

Manual de AngularJS



Alberto Basalo
Miguel Angel Alvarez
Xavier Jorge Cerdá
Pedro Hurtado



desarrolloweb.com/manuales/manual-polymer-2.html

Introducción: Manual de AngularJS

Este es un manual que nos introduce en el framework Javascript AngularJS, un conjunto de librerías de código abierto que nos sirven para hacer aplicaciones web avanzadas del lado del cliente. Es ideal para hacer aplicaciones de negocio y aplicaciones de gestión que se despliegan en una única página.

Usa el patrón de diseño habitualmente encontrado en el desarrollo web MVC, aunque en una variante llamada a veces MV* y a veces MVVM. Esto, junto con otras herramientas disponibles en Angular nos permite un desarrollo ordenado, sencillo de realizar y sobre todo más fácil de mantener en un futuro.

AngularJS está apoyado por Google y cada día más desarrolladores están adoptándolo, lo que nos da una idea del prometedor futuro de la librería.

Encuentras este manual online en:

<http://desarrolloweb.com/manuales/manual-angularjs.html>

Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

Miguel Angel Alvarez

Miguel es fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.



Alberto Basalo

Alberto Basalo es experto en Angular y otras tecnologías basadas en Javascript, como NodeJS y MongoDB. Es director de Ágora Binaria, empresa dedicada al desarrollo de aplicaciones y a la formación a través de Academia Binaria.



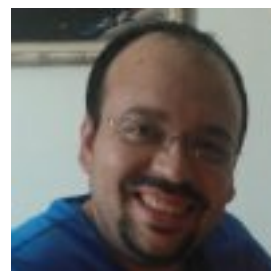
Pedro Hurtado

Amante de las novedades, defensor de la comunidad. Compartir es la fuente de la Sabiduría. No te cierres en tu mundo y da lo mismo tus creencias. Estamos en mundo abierto. Desarrollador en .Net, Nodejs, Angular convencido y porque no el resto, como G



Xavier Jorge Cerdá

Xavier, conocido entre sus amigos como Xavi Paper, es CTO en Ambiental Intelligence & Interaction, como desarrollador es especialista en tecnologías Microsoft y en Javascript, impulsor del framework AngularJS.



Introducción a AngularJS

Esta es una introducción de manera conceptual al framework Javascript AngularJS. Explicamos por qué cada vez tiene más importancia el desarrollo de aplicaciones web con alta carga de Javascript y por qué son esenciales este tipo de librerías para una programación más sencilla, rápida y de fácil mantenimiento. Completaremos la introducción con una práctica en la que podremos experimentar con un primer ejemplo en Angular.

Por qué AngularJS

Cómo los frameworks Javascript en general y AngularJS en concreto nos sirven para solucionar las necesidades actuales de la informática, en cuanto a desarrollo multiplataforma de aplicaciones grandes o enormes que se asemejan a las de escritorio.

Con este artículo comenzamos una serie dedicada a AngularJS, el framework Javascript MV* que nos permite un desarrollo rápido y potente de aplicaciones web basadas en Javascript del lado del cliente. AngularJS es un conjunto de librerías apoyadas por Google que tiene un futuro muy prometedor y que en estos momentos se encuentra en boca de todos. No se trata de una moda pasajera, sino de una tecnología que por su utilidad y características ha venido para quedarse.

A lo largo de varios artículos nos iremos introduciendo en este framework, de una manera bastante práctica. Sin embargo vamos a comenzar con una introducción más teórica y retrospectiva que nos haga entender el porqué de la proliferación de frameworks Javascript y el motivo por el que Angular es una apuesta segura entre sus competidores.



Nota: Esta es una transcripción libre de la charla de Alberto Basalo que nos ofreció en abierto en un webinar de DesarrolloWeb.com / EscuelaIT. El título de esta primera parte que ahora te resumimos es "Desarrollo de aplicaciones de negocio y los retos y soluciones de 2014". Al final del artículo podrás ver el vídeo de la charla.

Un poco de historia

"*El software sigue al hardware*". Esta es una afirmación que nos indica que programamos para aquellas máquinas en las que vamos a ejecutar los programas. Puede resultar obvio, pero como ejemplo se puede señalar que no vamos a programar para las "Google Glass" si aun no las han creado, o si aun no han liberado el SDK. Cuando empezó la informática en los años 60 existían ordenadores arcaicos y como programadores estabas limitado a las posibilidades de éstos.

Conforme avanzó el desarrollo de la informática aparecieron otros ordenadores. Al principio no estaban conectados entre sí ya que no existían las redes locales, ni mucho menos Internet. Como no había redes estabas limitado a lo que ocurría dentro de esa máquina y quizás los programadores tenían una vida más sencilla: estas limitaciones provocaban que no tuvieran que preocuparse por ciertas cosas. Incluso las opciones para crear los programas -tecnologías y lenguajes- no eran demasiadas, al contrario, quizás en tu sistema operativo estabas obligado a trabajar con un lenguaje o un par de ellos nada más.

Luego aparecieron las redes, apareció Internet y los ordenadores comenzaron a conectarse entre sí. Existen servidores y terminales que ya no son tontos, pero estamos trabajando con lenguajes sencillos, como HTML -al principio ni existía CSS- y ya acercándose al final del milenio aparecen lenguajes como Javascript capaces de hacer algunas cosas sencillas.

El reto hoy

Hoy la situación ya no es la que se describe anteriormente, sino que **el panorama ha cambiado mucho**. Por un lado el abaratamiento de las comunicaciones hace que aparezcan grandes centros de proceso de datos que nos facilitan el acceso a tecnología de primer orden, lo que se llama la nube. Ahora cualquier empresa, incluso las más pequeñas, tienen la posibilidad de acceder a servidores de aplicaciones.

Se ha acabado la "*tiranía de Windows*": hoy existen varios sistemas operativos y se usan para todo tipo de cosas. Los Mac ya no los usan solamente los diseñadores, los Linux no son terreno exclusivo de los desarrolladores, sino que cualquier persona usa esos sistemas para todo tipo de tareas. Los desarrolladores no pueden centrarse en un único sistema y limitar el servicio a las personas de determinada plataforma.

La cosa no queda ahí, puesto que ya no solo hablamos de ordenadores de escritorio y portátiles, además tenemos la tecnología móvil y los dispositivos como tablets. Yendo todavía un poco más allá tenemos los televisores inteligentes, las Google Glass y al final de todo, el "Internet of Things".

Esta es la foto actual, que es muy distinta a la de las últimas décadas. Tenemos la **nube con sus innumerables posibilidades** y tenemos una **infinita gama de ordenadores y dispositivos** a los que atender.

¿En qué programar?

A toda la situación relatada anteriormente le tenemos que sumar una interminable lista de necesidades en el campo de la informática y, por supuesto, tecnologías y lenguajes que nos sirven para resolverlas.

¿Programo para la web o para el escritorio? ¿realizo desarrollo nativo o multiplataforma? La industria te da soluciones y aporta todo tipo de alternativas, Java, .net, Python, RoR, Objective-C y otros más arcaicos como Cobol, pero es inevitable plantearse cuál de ellos es más adecuado para resolver los problemas.

HTML5 + JS

Si quieres realizar un desarrollo que se adapte a todo tipo de sistemas y dispositivos que puedan llegar a aparecer, una solución es buscar algo que sea común a todos los sistemas y buscar algo que tengas seguridad que lo van a entender todos.

Existe un **consenso en el mundo de la tecnología de dar soporte a HTML5 y Javascript**. La situación actual y la industria nos hace pensar que estos lenguajes estarán disponibles en todo sistema donde circule un bit. Por ello, podemos ver a estas tecnologías de estándares abiertos como un caballo ganador.

Eso sí, HTML5 + Javascript compiten con las soluciones nativas y en términos de rendimiento es muy difícil que puedan llegar a equipararse. El lenguaje nativo, además, siempre tendrá mayor facilidad de acceso a las características específicas y funcionalidades del dispositivo.

También compite con los lenguajes más clásicos como C, PHP, Java donde hay ya muchísimo trabajo ya realizado en forma de librerías disponibles para los programadores. Esta situación está cambiando, pero hay que observar que hasta hace poco era complicado hacer grandes programas con Javascript, pues el lenguaje servía para bien poco. Con la llegada de HTML5 y las diversas API se ha extendido mucho y se ha hecho mucho más poderoso, pero todavía faltaba mucho terreno para que Javascript se considerase un lenguaje robusto, capaz de cumplir las necesidades de aplicaciones grandes.

AngularJS y otros frameworks

Recientemente han aparecido una oleada de sistemas que han situado Javascript en otro nivel. AngularJS es uno de ellos, pero están otros muchos como BackboneJS o EmberJS. Son los frameworks que vienen a aportar herramientas y patrones de diseño con los que Javascript se convierte en un lenguaje capaz de servir como motor de aplicaciones enormes.

Y tiene todo el sentido que sea así. Hoy los ordenadores modernos, por muy modestos que sean, son capaces de procesar con velocidad ciertas cosas. Son capaces de recibir simples datos y "cocinarse" ellos mismos el HTML para visualizarlos a base de plantillas. Antes el servidor era el que tenía que enviar el HTML completo al cliente, ahora la tendencia es que solo envíe los datos y que el cliente (navegador o cualquier otro sistema donde desees ver esos datos) sea el que los trate y los muestre debidamente.

Esto ha producido que una parte de la lógica de maquetado y de presentación de la información se haya trasladado del servidor hacia los clientes. La ventaja obvia es que el servidor se ha descargado de trabajo, puesto que simplemente tiene que enviar los datos a través de JSON al cliente y es éste el que se encargará de producir el HTML que sea necesario. Pero no es solo una mejora en relación al servidor en términos de procesamiento, también en términos de bits, porque es más ligero transferir datos simples que el HTML completo para mostrarlos.

En definitiva, el servidor ha repartido la carga de trabajo que solía recaer sobre él entre todos los clientes que se conectan a su servicio. Pero la mejora no se queda solamente en el servidor, sino que el usuario también percibe un mejor desempeño, puesto que las acciones que realiza contra el servidor tienen una **respuesta más rápida**. Con ello poco a poco las aplicaciones cliente/servidor tienen un desempeño más parecido a las aplicaciones de escritorio. El usuario es el rey y demanda aplicaciones que sean rápidas y no le hagan esperar y eso se lo dan los frameworks como AngularJS.

Al programador además le facilitan las cosas, no solo por disponer de un conjunto de librerías, sino porque

los frameworks nos traen un conjunto de paradigmas y patrones que facilitan el desarrollo del software y sobre todo su mantenimiento. Nos referimos principalmente al llamado MVC, que es la separación del código en diferentes responsabilidades. Ahora cada parte del código debemos situarlo en un lugar determinado, y ese orden nos facilita que los desarrollos sean más manejables. Sobre todo esa mejora permite que en un futuro puedas "meter mano" al software para mantenerlo cuando sea necesario. Todo redundando en la **calidad del código**, lo que es indispensable para los proyectos y los programadores.

Potente, sencillo y extensible

Además hay un ecosistema de herramientas alrededor de los frameworks y no solo aquellas que están incorporadas en las librerías del propio AngularJS o cualquiera de sus competidores, sino una serie de servicios web y librerías de terceros que nos facilitan una enorme gama de objetivos.

Por poner dos ejemplos tenemos Apache Cordova, que es una librería para servir de puente a HTML5/JS hacia las características y funcionalidades de los dispositivos móviles. O sistemas como AngularFire que es un "backend as a service", que permite crear tu propia API y que ellos se ocupen de crear todos los sistemas para la persistencia de la información y la entrega de los JSON.

AngularJS mejora el HTML para crear aplicaciones web

AngularJS y otros frameworks tienen además la característica de mejorar el HTML existente, facilitando el desarrollo de aplicaciones. En este punto cabe recalcar la palabra "aplicaciones" puesto que este tipo de herramientas son adecuadas para realizar las llamadas "aplicaciones de gestión" o "aplicaciones de negocio".

Es importante esta mención porque AngularJS no es adecuado para resolver todo tipo de proyectos, o al menos no te facilitará especialmente ciertos desarrollos. Incluso por sus características habrá necesidades que ni siquiera sea adecuadas realizar en HTML5, como posiblemente un videojuego con gráficos avanzados, donde sería más adecuado una aplicación nativa (aunque esto en el futuro pueda cambiar).

Otro ejemplo es la realización de una aplicación intensiva de SEO. En cuanto a posicionamiento orgánico en buscadores el desarrollo con AngularJS, u otros frameworks Javascript, no es muy interesante porque el HTML que reciben los clientes -o los bots del motor de búsqueda- está prácticamente vacío de contenido y solo se rellena a posteriori por medio de solicitudes Ajax. Parece que Google está haciendo esfuerzos para que esta situación cambie y existen diversas soluciones a nivel de programación que pueden paliar en parte la carencia de SEO, pero lo cierto es que el desarrollo de la aplicación se complica al aplicarlas.

En fin, AngularJS nos ofrece muchas facilidades para hacer **aplicaciones web**, aplicaciones de gestión o de negocio, **aplicaciones que funcionan en dispositivos** y que tienen un rendimiento muy similar a las nativas e incluso **aplicaciones de escritorio con un frontal web**, cada vez más habituales.

Por qué Angular JS y no otros frameworks

En este sentido podría haber mucho que discutir entre partidarios de uno y otro framework, pero si dejamos a un lado las preferencias personales de cada uno, por aquella tecnología en la que haya apostado en el pasado, AngularJS es objetivamente mejor en muchos sentidos.

Primero y más importante es que con AngularJS requieres escribir menos código que con otros frameworks. Por ejemplo con respecto a BackboneJS hay muchas mejoras que son realmente críticas como el "doble

bindign" que te permite que los distintos componentes de tu aplicación estén al tanto de los cambios para modificar su estado automáticamente, sin necesidad de suscribirse a eventos y realizar otro tipo de acciones por medio de líneas de código. En este sentido hay tests objetivos que nos permiten ver que la misma aplicación hecha con AngularJS tiene sensiblemente menos código que en BackboneJS y quizás con otros frameworks pase lo mismo.

Segundo la comunidad. AngularJS tiene el apoyo de Google y una gran comunidad detrás. Las búsquedas de AngularJS se han disparado mientras que las de otros frameworks no han mejorado o han caído. Esto quiere decir en definitiva que encontrarás más documentación y más componentes de otros desarrolladores para basar tu trabajo en ellos.

Hasta aquí el primer artículo de AngularJS, que ha sido un tanto general. Solo esperamos que te hayas quedado con ganas de más. En el siguiente post te explicaremos qué es AngularJS.

A continuación encuentras la charla que ha servido de base para escribir este artículo, de Alberto Basalo. Muy entretenida e interesante, capaz de abrirnos la mente y motivarnos a aprender AngularJS.

Para ver este vídeo es necesario visitar el artículo original en: <http://desarrolloweb.com/articulos/por-que-angularjs.html>

Este artículo es obra de *Alberto Basalo*

Fue publicado por primera vez en 22/08/2014

Disponible online en <http://desarrolloweb.com/articulos/por-que-angularjs.html>

Qué es AngularJS

Descripción general sobre AngularJS, el framework Javascript, así como los principales componentes que tiene y los conceptos que debemos conocer antes de poner manos en el código.

AngularJS es Javascript. Es un proyecto de código abierto, realizado en Javascript que contiene un conjunto de librerías útiles para el desarrollo de aplicaciones web y propone una serie de patrones de diseño para llevarlas a cabo. En pocas palabras, es lo que se conoce como un framework para el desarrollo, en esta caso sobre el lenguaje Javascript con programación del lado del cliente.

Puedes encontrar el proyecto de AngularJS en su propio sitio web: [AngularJS, Superheroic JavaScript MVW Framework](#). Al ser un proyecto de código abierto cualquier persona con un poco de curiosidad echar un vistazo con profundidad y ver cómo se ha escrito, incluso admiten colaboraciones de desarrolladores que quiera aportar cosas.

Nota: Este artículo es una transcripción de la exposición de Alberto Basalo en DesarrolloWeb.com / EscuelaIT que se emitió en abierto por webinar. Al pie del texto encontrarás un vídeo de esta charla. El bloque anterior de esta ponencia la hemos publicado ya en el artículo ["Por qué AngularJS"](#).



Mejoras del HTML

Este Javascript pretende que los programadores mejoren el HTML que hacen. Que puedan producir un HTML que, de manera declarativa, genere aplicaciones que sean fáciles de entender incluso para alguien que no tiene conocimientos profundos de informática. El objetivo es producir un HTML altamente semántico, es decir, que cuando lo leas entiendas de manera clara qué es lo que hace o para qué sirve cada cosa.

Lógicamente, AngularJS viene cargado con todas las herramientas que los creadores ofrecen para que los desarrolladores sean capaces de crear ese HTML enriquecido. La palabra clave que permite ese HTML declarativo en AngularJS es "directiva", que no es otra cosa que código Javascript que mejora el HTML. Puedes usar el que viene con AngularJS y el que han hecho terceros desarrolladores, puesto que muchas personas están contribuyendo con pequeños proyectos -independientes del propio framework- para enriquecer el panorama de directivas disponibles. Hasta este punto serás un "consumidor de directivas", y finalmente cuando vayas tomando experiencia serás capaz de convertirte en un "productor de directivas", enriqueciendo tú mismo las herramientas para mejorar tu propio HTML.

Promueve patrones de diseño adecuados para aplicaciones web

Angular promueve y usa patrones de diseño de software. En concreto implementa lo que se llama MVC, aunque en una variante muy extendida en el mundo de Javascript que luego comentaremos con más detalle. Básicamente estos patrones nos marcan la separación del código de los programas dependiendo de su responsabilidad. Eso permite repartir la lógica de la aplicación por capas, lo que resulta muy adecuado para aplicaciones de negocio y para las aplicaciones SPA (Single Page Application).

Nota: Las SPA o "Aplicaciones de una sola página", son sitios web donde los usuarios perciben una experiencia similar a la que se tiene con las aplicaciones de escritorio. En este tipo de sitios la página no se recarga, no existe una navegación de una página a otra totalmente diferente, sino que se van intercambiando las "vistas". Técnicamente podríamos decir que, al interactuar con el sitio, el navegador no recarga todo el contenido, sino únicamente vistas dentro de la misma página.

AngularJS a vista de pájaro

Ahora vamos a hacer un breve recorrido para nombrar y describir con unos pequeños apuntes aquellos elementos y conceptos que te vas a encontrar dentro de AngularJS.

Primeramente tenemos que hablar sobre el gran patrón que se usa en Angular, el conocido Modelo, Vista,

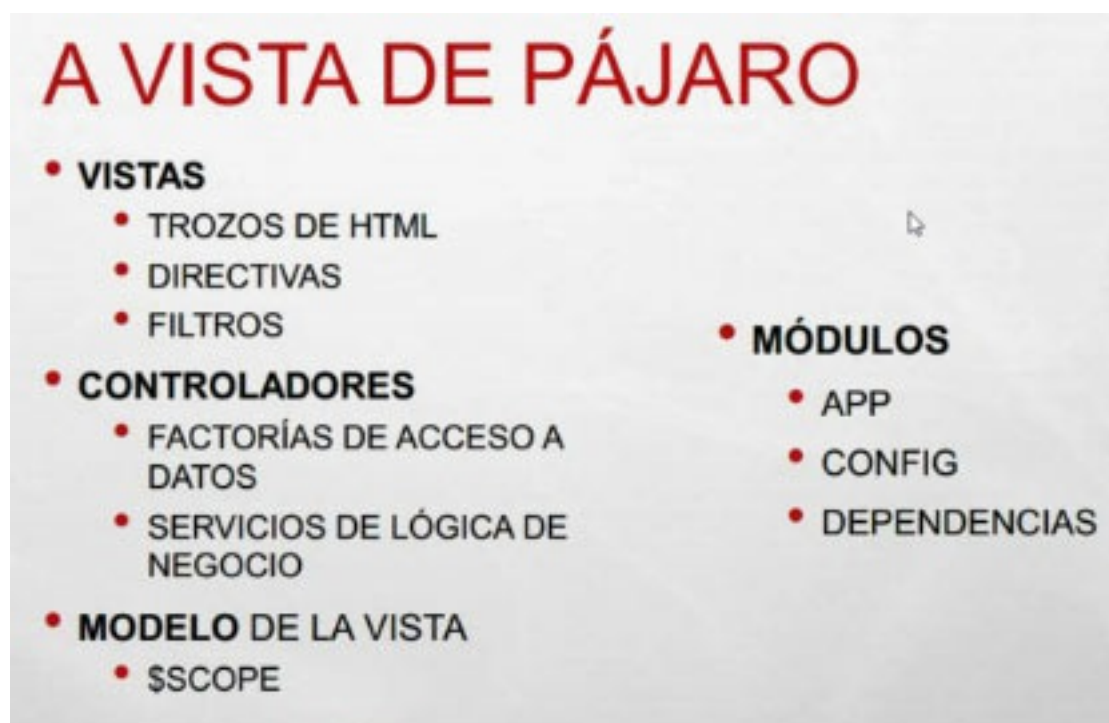
Controlador.

- **Vistas:** Será el HTML y todo lo que represente datos o información.
- **Controladores:** Se encargarán de la lógica de la aplicación y sobre todo de las llamadas "Factorías" y "Servicios" para mover datos contra servidores o memoria local en HTML5.
- **Modelo de la vista:** En Angular el "Modelo" es algo más de aquello que se entiende habitualmente cuando te hablan del MVC tradicional, osea, las vistas son algo más que el modelo de datos. En modo de ejemplo, en aplicaciones de negocio donde tienes que manejar la contabilidad de una empresa, el modelo serían los movimientos contables. Pero en una pantalla concreta de tu aplicación es posible que tengas que ver otras cosas, además del movimiento contable, como el nombre de los usuarios, los permisos que tienen, si pueden ver los datos, editarlos, etc. Toda esa información, que es útil para el programador pero que no forma parte del modelo del negocio, es a lo que llamamos el "Scope" que es el modelo en Angular.

Nota: Por ese motivo por el cual en AngularJS tienes unos modelos un poco diferentes, algunos autores dicen que el patrón que utiliza el framework es el MVVM Model-View-View-Model. En resumen, el modelo de la vista son los datos más los datos adicionales que necesitas para mostrarlos adecuadamente.

Además del patrón principal, descrito hasta ahora tenemos los módulos:

Módulos: La manera que nos va a proponer AngularJS para que nosotros como desarrolladores seamos cada vez más ordenados, que no tengamos excusas para no hacer un buen código, para evitar el código espagueti, ficheros gigantescos con miles de líneas de código, etc. Podemos dividir las cosas, evitar el infierno de las variables globales en Javascript, etc. Con los módulos podemos realizar aplicaciones bien hechas, de las que un programador pueda sentirse orgulloso y sobre todo, que nos facilite su desarrollo y el mantenimiento.



Dos "mundos" en AngularJS

Ahora tenemos que examinar AngularJS bajo otra perspectiva, que nos facilite entender algunos conceptos y prácticas habituales en el desarrollo. Para ello dividimos el panorama del framework en dos áreas.

- **Parte del HTML:** Es la parte declarativa, con las vistas, así como las directivas y filtros que nos provee AngularJS, así como los que hagamos nosotros mismos o terceros desarrolladores.
- **Parte Javascript puro:** Que serán los controladores, factorías y servicios.



Es importante señalar aquí, aunque se volverá a incidir sobre ese punto, que nunca jamás se deberá acceder al DOM desde la parte del Javascript. Es un pecado mortal ya que esa parte debe ser programada de manera agnóstica, sin tener en cuenta la manera en la que se van a presentar los datos.

En medio tendremos el denominado Scope, que como decimos representa al modelo en Angular. En resumen no es más que un objeto Javascript el cual puedes extender creando propiedades que pueden ser datos o funciones. Nos sirve para comunicar desde la parte del HTML a la parte del Javascript y viceversa. Es donde se produce la "magia" en Angular y aunque esto no sea del todo cierto, a modo de explicación para que se entienda algo mejor, podemos decir que AngularJS se va a suscribir a los cambios que ocurran en el scope para actualizar la vista. Y al revés, se suscribirá a los cambios que ocurran en la vista y con eso actualizará el scope.

En el siguiente artículo comenzaremos ya con código, así que los impacientes tendrán ya sus deseos hechos realidad.

Puedes ver el vídeo de esta parte de la presentación sobre Qué es AngularJS.

Para ver este vídeo es necesario visitar el artículo original en: <http://desarrolloweb.com/articulos/que-es-angularjs-descripcion-framework-javascript-conceptos.html>

Este artículo es obra de *Alberto Basalo*

Fue publicado por primera vez en 28/08/2014

Disponible online en <http://desarrolloweb.com/articulos/que-es-angularjs-descripcion-framework-javascript-conceptos.html>

AngularJS Vs jQuery

¿Complementarias? ¿Competidoras? Analizamos similitudes y diferencias de jQuery y AngularJS, casos en los que sería más aconsejable una u otra librería y una comparativa de código, de un mismo problema, resuelto con ambas herramientas.

No quiero comenzar a escribir sin explicar bien el título de este artículo. Quien tenga alguna idea de AngularJS quizás piense que hemos patinado al intentar enfrentarlo a jQuery pues son librerías con filosofías diferentes, que sirven para resolver distintos tipos de problemas. En definitiva, las dos son para endulzar el Javascript del lado del cliente, pero cada una es adecuada para un tipo de problemas. Sobre todo esto hablaremos con bastante detalle a lo largo del artículo, para no causar confusión entre los lectores, tanto noveles como experimentados. Unimos además una comparativa de las filosofías de ambas herramientas, reflejada en el código de resolución de un mismo problema, lo que nos ayudará a entender mejor sus diferencias.

Dicho eso, aclaro que la elección del título responde a ésta es una duda habitual de las personas que se interesan por Angular u otras librerías MV* de Javascript. Si tú tienes la duda, o quieres entender mejor la filosofía de AngularJS, puedes seguir leyendo. Si tú ya conoces perfectamente las diferencias y no necesitas saber más, puedes saltarte este artículo y seguir tranquilamente por el siguiente del [Manual de AngularJS](#).



Muchos de nosotros hemos entrado en el mundo de las librerías Javascript de la mano de jQuery. Incluso hay también muchos desarrolladores que han entrado en el lenguaje Javascript aprendiendo jQuery. El caso es que en el pasado lustro jQuery ha dominado el mundo del desarrollo del lado del cliente y sigue usándose de manera muy habitual, hasta el punto de considerarse casi un estándar. Sin embargo, nuevas librerías Javascript y frameworks como AngularJS van un paso más allá de donde se queda jQuery, aportando muchas otras utilidades y patrones de diseño de software.

Después de haber invertido tanto tiempo en el aprendizaje de jQuery, una de las preguntas que muchos nos hacemos cuando conocemos AngularJS es ¿Qué me queda de todo eso cuando me pase al desarrollo con AngularJS? ¿Qué pasa con jQuery? ¿Sirven para hacer lo mismo? ¿son competidores? ¿complementarios?

Nota: Parte de este artículo es transcripción de una parte de la presentación sobre AngularJS que nos ofreció en DesarrolloWeb.com Alberto Basalo. Además adicionalmente hemos añadido una comparativa entre el "Hola mundo" típico de AngularJS con esta librería y el código que deberías emplear si usas jQuery.

AngularJS y jQuery son dos librerías de alcances distintos

Si queremos entrar en esta discusión, y para no liar a aquellos desarrolladores con menos experiencia, debemos decir que jQuery y AngularJS son librerías bien diferentes. El alcance y el tipo de cosas que se hacen con una y otra librería son distintos.

jQuery es una librería que nos sirve para acceder y modificar el estado de cualquiera de los elementos de la página. A través de jQuery y los selectores de CSS (así como los selectores creados por el propio jQuery) eres capaz de llegar a los elementos de la página, a cualquiera de ellos, y puedes leer y modificar sus propiedades, suscribirte a eventos que ocurran en esos elementos, etc. Con jQuery podíamos manejar cualquier cosa que ocurra en esos elementos de una manera mucho más cómoda que con Javascript "a pelo" y compatible con la mayor gama de navegadores.

Sin embargo Angular pasa de ser una librería para convertirse en un framework de aplicaciones web. No solo te permite una serie de funciones y mecanismos para acceder a los elementos de la página y modificarlos, sino que también te ofrece una serie de mecanismos por los cuales extender el HTML, para hacerlo más semántico, incluso ahorrarte muchas líneas de código Javascript para hacer las mismas cosas que antes hacías con jQuery. Pero la principal diferencia y por la cual AngularJS toma la denominación de "framework", es que te marca una serie de normas y hábitos en la programación, principalmente gracias al patrón MVC implementado en AngularJS.

Nota: Para más información puedes leer los capítulos iniciales del [Manual de AngularJS](#), donde aprenderás más sobre las características del framework.

jqLite dentro de AngularJS

"AngularJS se ha comido a jQuery". Dentro de AngularJS tienes una pequeña implementación de jQuery. Todos hemos usado millones de veces jQuery y es una librería excelente porque nos facilita el acceso al DOM y además nos ofrece un API de funciones que son compatibles con todos los navegadores.

Eso mismo lo tienes dentro del script de AngularJS, por lo que si no lo deseas, no necesitas usar jQuery para nada. Dentro de Angular tienes jqLite "jQuery Lite" que viene a ser una librería de acceso al DOM, con la mínima funcionalidad necesaria.

Esto quiere decir que las directivas que trae AngularJS usan jQuery por debajo, o algo muy similar, un subconjunto de jQuery. Por supuesto, cuando tú programes tus propias directivas puedes usar la librería de acceso al DOM que viene con AngularJS. Si esa pequeña librería no te resulta suficiente y necesitas mayor funcionalidad, nadie te impide usar el jQuery de verdad.

Si AngularJS detecta que estás usando jQuery, todo el tema de acceso al DOM lo hará a través de jQuery, dejando su implementación (jqLite) sin usar. Si detecta que no tienes jQuery, entonces pone en marcha su propia librería de acceso al DOM.

Nota: Incluso jQuery puede traer mejoras de rendimiento con respecto a la misma funcionalidad que viene en AngularJS. No es el caso ahora estudiar el rendimiento de una y otra librería sino entender sus

diferencias. No obstante, a pesar de jQuery poder dar un poco más de rendimiento en tiempo de ejecución, se notará solamente con trabajos intensivos y habría que tener en cuenta el tiempo de carga de la librería adicional para comparar definitivamente usar Angular a secas o Angular con jQuery.

Entonces uso jQuery dentro de AngularJS ¿sí o no?

Sí, puedes usarlo, aunque no es absolutamente necesario.

Sin embargo, debe quedar claro que el uso de jQuery debe quedar restringido a las vistas, filtros y creación de nuevas directivas, nunca al uso de factorías, servicios o controladores. Cuando una persona que viene de jQuery y comienza a usar AngularJS es normal que caiga en costumbres y mecanismos a los que está habituado con aquella librería. Por ejemplo, se corre el riesgo de querer acceder al DOM desde los controladores y es algo que no se debe realizar bajo ningún concepto.

En ese sentido, para las personas que usan AngularJS y están acostumbradas a jQuery, en un principio al menos puede ser recomendable evitar siempre que se pueda usar jQuery. Es decir, "no usar jQuery hasta que no sea estrictamente necesario". Y cuando lo uses, insístimos, siempre será desde la definición de las directivas, nunca desde los propios controladores.

Si tus requisitos son más sencillos, realización de sitios web en los que tienes poco peso de Javascript y limitado a crear interfaces de usuario dinámicas o pequeños comportamientos interactivos, puedes seguir usando jQuery. AngularJS está más destinada a aplicaciones web complejas, más parecidas a las aplicaciones de escritorio, con uso muy intensivo de Javascript. No todos los problemas típicos de un sitio web sería adecuado resolver con AngularJS, pues a veces será como "matar moscas a cañonazos" y por la complejidad de Angular te puede generar código más difícil de producir y sobre todo de entender para una persona con menos experiencia.

Además el hecho de Angular usar patrones de diseño como el MVC puede despistar bastante a las personas con menos bagaje en el mundo de la programación. Los patrones de diseño son algo que está para ayudarnos, pero no todos los proyectos y desarrolladores los necesitan, pues complica la curva de aprendizaje de las tecnologías y en ocasiones si nuestro problema es sencillo no ofrecen una solución sensiblemente mejor.

Comparativa de código jQuery / AngularJS

Ha quedado claro que Angular va más allá del objetivo de jQuery. Son dos librerías distintas, que muchas veces se pueden complementar, pero dependiendo del tipo de proyecto y de las cosas que tengas que hacer con Javascript elegirás una u otra. No obstante, queremos hacer esta comparativa de código para observar algunas de las diferencias de filosofía entre jQuery y AngularJS y también comprobar de paso la potencia de este último.

En este caso vamos a comparar el código que vimos en el [Hola Mundo de AngularJS](#). La idea es simplemente escribir un nombre en un campo de texto y volcarlo en una etiqueta H1.

Esto en jQuery se podría hacer con un código como este:

```
<!DOCTYPE html>
```



```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Ej Hola Mundo desarrollado en jQuery</title>
</head>
<body>
  <h1>Hola</h1>
  <form>
    ¿Cómo te llamas? <input type="text" id="nombre">
  </form>

  <script src="https://code.jquery.com/jquery-1.11.1.min.js"></script>
  <script>
    $(function(){
      var campoTexto = $("#nombre");
      var titular = $("h1");
      campoTexto.on("keyup", function(){
        titular.text("Hola " + campoTexto.val());
      });
    });
  </script>
</body>
</html>
```

En AngularJS ya vimos el código necesario en un artículo anterior, pero lo reproducimos para poder compararlos fácilmente.

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
  <meta charset="UTF-8">
  <title>Ej Hola Mundo desarrollado en AngularJS</title>
</head>
<body>
  <h1>Hola {{nombre}}</h1>
  <form>
    ¿Cómo te llamas? <input type="text" ng-model="nombre">
  </form>

  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.min.js"></script>
</body>
</html>
```

Nota: Esta comparativa la puedes ver realizada en vídeo, en inglés, aunque resulta realmente fácil de seguirlo si entiendes un poco estos códigos: <https://www.youtube.com/watch?v=uFTFsKmkQnQ>

Como habrás observado, con AngularJS se simplifica nuestra vida. No solo que no necesitas hacer muchas de las cosas que en jQuery tienes que programar "a mano", como el acceso al campo de texto para recuperar

el valor, la creación del evento de teclado para hacer cosas cuando se escribe dentro del campo de texto, etc. Lo que más debería llamar nuestra atención es que el código de AngularJS es mucho más entendible, incluso por personas que no tengan idea de programar. Como puedes deducir, en Angular escribimos generalmente menos código Javascript, pero sí es importante decir que debido al uso del patrón MV* (separación del código por sus diferentes objetivos), en ejemplos más avanzados observarás que entran en juego diferentes factores que complican las soluciones. Generalmente, aunque en la métrica de líneas de código AngularJS pueda ganar a jQuery, también debes saber que te exigirá mayor capacidad de abstracción y entender diversos conceptos de patrones de diseño de software.

Todo ello nos debe hacer entender mejor la potencia de AngularJS. Pero ojo, sin desmerecer a jQuery, pues debe quedar claro que cada librería es útil en su campo.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 16/10/2014
Disponible online en <http://desarrolloweb.com/articulos/angularjs-vs-jquery.html>

Primeros pasos con AngularJS

Qué necesitas para trabajar con AngularJS, como descargar el framework y cómo realizar un primer programa, el típico Hola Mundo.

Ya hemos conocido por qué AngularJS nos ayuda en el desarrollo de sitios web modernos y también hemos explorado en detalle diversos conceptos iniciales para manejar esta librería, así que estamos en condiciones de poner manos en el código.

En este artículo queremos ofrecer una introducción muy básica a AngularJS, creando un extraordinariamente sencillo "Hola mundo" con el que podremos empezar a apreciar la potencia de este framework Javascript. Los impacientes e manejar código estaréis satisfechos después de la lectura de este artículo, pero antes que nada tenemos que saber qué materiales necesito en mi ordenador para comenzar a trabajar.



Qué necesitas para desarrollar con AngularJS

Lo único que necesitas para desarrollar con AngularJS es un editor de texto y un navegador. Así de sencillo, seguramente todo el mundo tenga ya configurado su editor preferido, así que te puedes saltar si quieres estos siguientes párrafos. Para los que no tengan claro esto, os dejo un par de comentarios.

- **Editor de código:** puede ser cualquiera que estés acostumbrado a usar, no necesitas cambiarlo para programar con Angular. Lo que es interesante es que sea un editor para programadores, que te permita diversas ayudas al escribir el código, coloreado de sintaxis, ayudas contextuales, etc. Notepad++, Sublime Text, Komodo Edit, Brackets, etc. Cualquiera es bueno. Si estás acostumbrado a IDEs más complejos como Eclipse o PhpStorm, puedes quedarte también con ellos.
- **Navegador:** Puedes usar cualquier navegador para ver un desarrollo basado en Angular. Generalmente tendrás incluso varios navegadores para probar tu página en cada uno de ellos y comprobar que todo está en orden. Solo te recomendamos tener a Google Chrome entre tu batería de navegadores, pues luego hablaremos de una extensión que existe para éste que te permite examinar y depurar páginas donde AngularJS está trabajando.

Descarga de AngularJS

Si quieres trabajar con AngularJS tienes que incluir el script del framework en tu página. Esto lo puedes hacer de varias maneras, o bien te descargas la librería por completo y la colocas en un directorio de tu proyecto, o bien usas un CDN para traerte la librería desde un servidor remoto. En principio es indiferente a nivel didáctico, así que nosotros vamos a comenzar del modo más sencillo, que es utilizar el CDN.

Accedes a la página de AngularJS: <https://angularjs.org/>

Pulsas el botón de descarga y encontrarás diversas opciones. Escoges la versión del framework (si es que te lo permite) y que esté minimizada (minified). Luego encontrarás un campo de texto donde está la URL de la librería (esa URL está marcada con las siglas "CDN" al lado). Ya sabes que el CDN te ofrece un contenido, en este caso un script Javascript, que está alojado en otro servidor, pero que lo puedes usar desde tu página para mejorar la entrega del mismo.

Nota: La versión minimizada pesa menos en Kb, por lo que será la adecuada para cualquier sitio que tengas en producción. Sin embargo la versión sin comprimir "Uncompressed" tiene el código completo, con comentarios, sangrado, etc. lo que hace que sea más fácil de leer por humanos. Puedes usar si lo deseas la versión sin comprimir en la etapa de desarrollo.

Será algo como <https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.min.js> aunque todo depende de la versión que haya en el momento en el que tú leas esta guía de iniciación.

Incluir AngularJS en una página web

Una vez tienes tu CDN puedes incluir el script de Angular en la página con la etiqueta SCRIPT. Ese script lo puedes colocar en el HEAD o bien antes del final del BODY, en principio no habría diferencias en lo relativo a la funcionalidad, pero sí hay una pequeña mejora si lo colocas antes de cerrar el cuerpo.

Simplemente, si lo colocas en el HEAD estás obligando a que tu navegador se descargue la librería de AngularJS, retrasando quizás la descarga de áreas de la página con contenido. Si lo colocas antes de cerrar el BODY facilitas la vida a tu navegador, y por añadido a tus usuarios, pues podrá descargar todo el HTML, ir renderizando en la pantalla del usuario los contenidos sin entretenerse descargando AngularJS hasta que sea

realmente necesario.

Declarar directivas

Hay un paso más para dejar lista una página donde quieras usar AngularJS. Es simplemente colocar la directiva `ng-app` en la etiqueta que englobe la aplicación. Más adelante hablaremos con más detalle de las directivas y daremos algunos tips para usarlas mejor. Por ahora puedes quedarte simplemente con la necesidad de informar a AngularJS del contenedor HTML donde va a desplegar su "magia".

Típicamente pondrás `ng-app` en la etiqueta HTML de inicio del documento.

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
<meta charset="UTF-8">
<title>Ej de AngularJS</title>
</head>
<body>

... Aquí el cuerpo de tu página ...

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.min.js"></script>
</body>
</html>
```

Así como `ng-app`, existen muchas otras directivas para enriquecer tu HTML, aun veremos alguna más en este artículo.

Hola Mundo en AngularJS

Ahora vamos a poner algo de carne en el asador y vamos a observar la potencia de AngularJS con muy poco código. Realmente, como observarás, se trata de cero código (Javascript).

Nota: El código de este Hola Mundo lo puedes encontrar también, prácticamente la misma implementación, en la home de AngularJS.

Vamos a colocar un formulario con un campo de texto.

```
<form>
¿Cómo te llamas? <input type="text" ng-model="nombre">
</form>
```

Observa que en el campo de texto hemos usado `ng-model` y le hemos asignado un valor. Ese `ng-model` es otra directiva que nos dice que ese campo de texto forma parte de nuestro modelo y el valor "nombre" es la referencia con la que se conocerá a este dato. Insisto en que más adelante hablaremos con detalle sobre

estos datos y veremos nuevos ejemplos.

Ahora vamos a crear un elemento de tu página donde volcaremos lo que haya escrito en ese campo de texto.

```
<h1>Hola {{nombre}}</h1>
```

Como ves, dentro del H1 tenemos `{{nombre}}`. Esas dobles llaves nos sirven para indicarle a AngularJS que lo que hay dentro es una expresión. Allí podemos colocar cosas (código) para que Angular resuelva por nosotros. En este caso estamos colocando simplemente el nombre del modelo o dato que queremos mostrar.

Código completo

Nuestro ejemplo es perfectamente funcional. Si estabas esperando que escribiésemos algo de Javascript, lamento decepcionarte. La potencia de AngularJS es justamente que muchas cosas se pueden hacer en el documento HTML simplemente extendiendo sus posibilidades a través de directivas y expresiones.

Claro que para que esto funcione, por debajo el framework hace muchas cosas, pero de momento no es el caso extenderse más.

Puedes ver el código completo a continuación. Puedes copiarlo y pegarlo tal cual en tu editor de texto, en un nuevo archivo .html y debería de funcionar sin ningún problema.

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
  <meta charset="UTF-8">
  <title>Ej de AngularJS</title>
</head>
<body>
  <h1>Hola {{nombre}}</h1>
  <div class="contenedor">
    <form action="">
      ¿Cómo te llamas? <input type="text" ng-model="nombre">
    </form>
  </div>

  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.min.js"></script>
</body>
</html>
```

En el siguiente artículo nos extenderemos un poco más en las explicaciones. De momento es interesante que observes que sin necesidad de escribir código Javascript AngularJS es capaz de implementar una bonita funcionalidad. Solo piensa el código que necesitarías con Javascript "nativo" para hacer este ejercicio o usando otros sistemas como el popular jQuery.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 04/09/2014
Disponible online en <http://desarrolloweb.com/articulos/primeros-pasos-angularjs.html>

Binding en AngularJS, y doble binding

El enlace de datos es una de las cosas que facilita la programación con AngularJS, en inglés **Data binding**, siendo que en Angular tenemos un doble binding.

Este artículo es solamente para repasar, y poner los puntos sobre las íes, un concepto que ya hemos visto a lo largo del [Manual de AngularJS](#), pero que no hemos tratado de manera aislada. Se trata del data binding, uno de los puntos fuertes de Angular y donde reside la "magia" del framework.

El binding es algo que conoces ya en Javascript común, e incluso en librerías como jQuery, aunque es posible que no le has dado ese nombre todavía. La traducción de "binding" sería "enlace" y sirve para eso justamente, realizar un nexo de unión entre unas cosas y otras. Data binding sería "enlace de datos".

Además, una de las características de AngularJS es producir automáticamente lo que se llama el "doble binding" que nos facilita enormemente nuestro trabajo por ahorrarnos muchas líneas de código para realizarlo a mano. El doble binding no existe en todos los frameworks MVC de Javascript como BackboneJS, donde solo tenemos un "bindeo" simple.



Qué es el binding

El binding no es más que enlazar la información que tenemos en el "scope" con lo que mostramos en el HTML. Esto se produce en dos sentidos:

One-way binding:

En este caso la información solamente fluye desde el scope hacia la parte visual, osea, desde el modelo hacia la representación de la información en el HTML. Lo conseguimos con la sintaxis "Mustache" de las dos llaves.

```
{{ dato }}
```

Ese dato estarías trayéndolo desde el scope y mostrándolo en la página. La información fluye desde el scope hacia la representación quiere decir que, si por lo que sea se actualiza el dato que hay almacenado en el modelo (scope) se actualizará automáticamente en la presentación (página).

Two-way binding:

En este segundo caso la información fluye desde el scope hacia la parte visual (igual que en "one-way binding") y también desde la parte visual hacia el scope. La implementas por medio de la directiva ngModel.

```
<input type="text" ng-model="miDato" />
```

En este caso cuando el modelo cambie, el dato que está escrito en el campo de texto (o en el elemento de formulario donde lo uses) se actualizaría automáticamente con el nuevo valor. Además, gracias al doble binding (two-way) en este caso, cuando el usuario cambie el valor del campo de texto el scope se actualizará automáticamente.

Cómo puedo experimentar el binding

Realmente en los ejemplos más sencillos de AngularJS ya hemos experimentado el binding. No me hace falta ni Javascript para poder hacer un ejemplo donde se pueda ver en marcha perfectamente.

```
<div ng-app>
  <input type="text" ng-model="dato" />
  {{dato}}
  <input type="button" value="hola" ng-click="dato='hola'" />
</div>
```

Mira, tienes tres elementos destacables.

- Un campo de texto con ng-model="dato". En ese campo tenemos un "doble binding".
- Una expresión {{dato}} donde se mostrará aquello que haya escrito en el campo de texto. En este caso tenemos un binding simple, de un único sentido.
- finalmente tienes un botón que no tiene binding alguno. Simplemente, cuando lo pulses, estás ejecutando una expresión (con ng-click) para cambiar el scope en su variable "dato".

El binding one-way lo ves muy fácilmente. Simplemente escribe algo en el campo de texto y observa como se actualiza el lugar de la página donde estás volcándolo el valor con {{dato}}.

Para poder ver bien el doble binding he tenido que crear el botón, que hace cambios en el scope. Aclaro de nuevo que el doble binding de todos modos lo tienes en el INPUT text. Dirección 1) lo ves cuando escribes datos en el campo de texto, que viajan al modelo automáticamente (y sabemos que es cierto porque vemos cómo {{dato}} actualiza su valor. Dirección 2) lo ves cuando haces click en el botón. Entonces se actualiza ese dato del modelo y automáticamente viaja ese nuevo valor del scope hacia el campo de texto.

Por qué el binding es tan útil

Piensa en una aplicación que realizases con Javascript común (sin usar librería alguna), o incluso con jQuery que no tiene un binding automático. Piensa todo el código que tendrías que hacer para implementar ese pequeño ejemplo. Suscribirte eventos, definir las funciones manejadoras para que cuando cambie el estado del campo de texto, volcarlo sobre la página. Crear el manejador del botón para que cuando lo pulses se envíe el nuevo texto "hola" tanto al campo de texto como a la página, etc.

No estamos diciendo que sea difícil hacer eso "a mano", seguro que la mayoría es capaz de hacerlo en jQuery, pero fíjate que para cada una de esas pequeñas tareas tienes que agregar varias líneas de código, definir eventos, manejadores, especificar el código para que los datos viajen de un lugar a otro. Quizás no es tan complicado, pero sí es bastante laborioso y en aplicaciones complejas comienza a resultar un infierno tener que estar pendiente de tantas cosas.

Otra ventaja, aparte del propio tiempo que ahorras, es una limpieza de código muy destacable, ya que no tienes que estar implementando muchos eventos ni tienes necesidad de enviar datos de un sitio a otro. Tal como te viene un JSON de una llamada a un servicio web lo enganchas al scope y automáticamente lo tienes disponible en la vista, lo que es una maravilla. Esto se ve de manera notable en comparación con otros frameworks como BackboneJS.

Nota: Osea que el doble binding de Angular nos ahorra mucho trabajo, pero ojo, quizás no sea necesario para cualquier tipo de aplicación. Si tus necesidades no son muy grandes igual no consigues adelantar tanto. Incluso en aplicaciones con inmensa cantidad de datos puede que el doble binding te pueda ralentizar un poco la aplicación porque toda esa gestión del binding se haga pesada, igual de manera innecesaria. En esos casos quizás otra librería funcione mejor que AngularJS. Sin embargo, la inmensa mayoría de las aplicaciones se beneficiarán de ese doble binding.

Dónde encontrarás el binding

El binding que dispones en AngularJS se implementa automáticamente en muchas otras directivas que existen en el framework.

Es muy útil cuando trabajamos con colecciones y directivas como ngOptions o ngRepeat, que nos sirven para realizar recorridos a esas colecciones para crear los campos OPTION de un SELECT o simplemente para repetir cualquier pedazo del DOM. En estos casos, cuando la colección cambia, nuestra página se actualiza automáticamente.

También puedes hacer binding a atributos de diversas etiquetas de la página, con las directivas ngHref o ngSrc, ngStyle, ngClass. También funciona en formularios con una especie de framework de validación "Validations" o con el propio ngSubmit. Todo eso lo iremos viendo más adelante con ejemplos.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 20/10/2014
Disponible online en <http://desarrolloweb.com/articulos/binding-angularjs-doble-binding.html>

Desarrollo declarativo con AngularJS

Comenzamos a introducirnos en el mundo de Angular con una serie de artículos básicos dedicados a lo que se conoce como el HTML declarativo, una de las características de esta librería, que nos permite realizar aplicaciones sencillas sin necesidad de Javascript. El proceso consiste en crear una serie de declaraciones en el propio lenguaje de marcación en las que se indica qué cometido tienen los elementos de la página dentro de nuestra aplicación web.

Directivas y expresiones en AngularJS

Una primera aproximación a los componentes que forman parte de las aplicaciones básicas que podremos desarrollar con AngularJS, las directivas y las expresiones.

Hemos visto que Angular tiene como característica que extiende el HTML, pudiendo llegar a programar funcionalidad de la aplicación sin necesidad de escribir código Javascript. Ahora vamos a ver de manera más detallada algunos de los componentes típicos que encontramos en las aplicaciones desarrolladas con esta potente librería.

Al trabajar con AngularJS seguimos desarrollando encima del código HTML, pero ahora tenemos otros componentes útiles que agregan valor semántico a tu aplicación. De alguna manera estás enriqueciendo el HTML, por medio de lo que se conoce como "directiva".



Qué son las directivas

Las directivas son nuevos "comandos" que vas a incorporar al HTML y los puedes asignar a cualquiera de las etiquetas por medio de atributos. Son como marcas en elementos del DOM de tu página que le indican a AngularJS que tienen que asignarles un comportamiento determinado o incluso transformar ese elemento del DOM o alguno de sus hijos.

Cuando se ejecuta una aplicación que trabaja con Angular, existe un "HTML Compiler" (Compilador HTML) que se encarga de recorrer el documento y localizar las directivas que hayas colocado dentro del código HTML, para ejecutar aquellos comportamientos asociados a esas directivas.

AngularJS nos trae una serie de directivas "de fábrica" que nos sirven para hacer cosas habituales, así como tú o terceros desarrolladores pueden crear sus propias directivas para enriquecer el framework.

Directiva ngApp (ng-app)

Esta es la marca que indica el elemento raíz de tu aplicación. Se coloca como atributo en la etiqueta que deseas que sea la raíz. Es una directiva que autoarranca la aplicación web AngularJS. Se leería "Enyi ap" y lo más común es ponerlo al principio de tu documento HTML, en la etiqueta HTML o BODY, pero también lo podrías colocar en un área más restringida dentro del documento en otra de las etiquetas de tu página.

```
<html ng-app>
```

Nota: Como puedes comprobar, ng-app es una directiva y se indica como si fuera un atributo el HTML. Pero no lo es ningún atributo real de HTML. Si usas un validador de HTML te advertirá que ese atributo es inexistente y se interpretará por un error de validación. Para solucionar este posible inconveniente, la práctica aconsejable es colocar el prefijo "data-" a cada directiva.

```
<html data-ng-app>
```

De ese modo tu código validará perfectamente, ya que en HTML5 se pueden crear cualquier tipo de atributos personalizados que comiencen por "data-" y asignarles cualquier tipo de dato que queramos almacenar en la etiqueta.

Para no causar confusiones, también podemos agregar que a nivel de Javascript las directivas las encontrarás nombradas con notación "camel case", algo como ngApp. En la documentación también las encuentras nombradas con camel case, sin embargo, como el HTML no es sensible a las mayúsculas y minúsculas no tiene tanto sentido usar esa notación y por ello se separan las palabras de las directivas por un guión "-".

Opcionalmente ngApp puede contener como valor un módulo de AngularJS a cargar. Esto lo veremos más adelante cuando trabajemos con módulos.

Directiva ngModel (ng-model)

La directiva ngModel informa al compilador HTML de AngularJS que estás declarando una variable de tu modelo. Las puedes usar dentro de campos INPUT, SELECT, TEXTAREA (o controles de formulario personalizados).

Nota: De nuevo, cabe decir que no forma parte del estándar HTML sino que es una extensión que tenemos gracias a AngularJS.

La indicas con el atributo del HTML ng-model, asignando el nombre de la variable de tu modelo que estás declarando.

```
<input type="text" ng-model="busqueda">
```

Con eso le estás diciendo al framework que esté atento a lo que haya escrito en ese campo de texto, porque es una variable que vas a utilizar para almacenar algo y porque es importante para tu aplicación.

Técnicamente, lo que haces con ngModel es crear una propiedad dentro del "scope" (tu modelo) cuyo valor tendrá aquello que se escriba en el campo de texto. Gracias al "binding" cuando modifiques ese valor en el scope por medio de Javascript, también se modificará lo que haya escrito en el campo de texto. Aunque todo esto lo experimentarás y entenderás mejor un poco más adelante cuando nos metamos más de lleno en los controladores.

Expresiones

Con las expresiones también enriquecemos el HTML, ya que nos permiten colocar cualquier cosa y que AngularJS se encargue de interpretarla y resolverla. Para crear una expresión simplemente la englobas dentro de dobles llaves, de inicio y fin.

Ahora dentro de tu aplicación, en cualquier lugar de tu código HTML delimitado por la etiqueta donde pusiste la directiva ng-app eres capaz de colocar expresiones. En Angular podemos una gama de tipos de expresiones, de momento hagamos una prueba colocando una expresión así.

```
<h1>{{ 1 + 1 }}</h1>
```

Angular cuando se pone a ejecutar la aplicación buscará expresiones de este estilo y lo que tengan dentro, si es algo que él pueda evaluar, lo resolverá y lo sustituirá con el resultado que corresponda.

Puedes probar otra expresión como esta:

```
{{ "Hola " + "DesarrolloWeb" }}
```

Al ejecutarlo, AngularJS sustituirá esa expresión por "Hola DesarrolloWeb". Estas expresiones no me aportan gran cosa de momento, pero luego las utilizaremos para más tipos de operaciones.

Lo habitual de las expresiones es utilizarlas para colocar datos de tu modelo, escribiendo los nombres de las "variables" que tengas en el modelo. Por ejemplo, si tienes este código HTML, donde has definido un dato en tu modelo llamado "valor":

```
<input type="text" ng-model="valor" />
```

Podrás volcar en la página ese dato (lo que haya escrito en el campo INPUT) por medio de la siguiente expresión:

```
{{valor}}
```

Otro detalle interesante de las expresiones es la capacidad de formatear la salida, por ejemplo, diciendo que lo que se va a escribir es un número y que deben representarse dos decimales necesariamente. Vemos rápidamente la sintaxis de esta salida formateada, y más adelante la utilizaremos en diversos ejemplos:

```
{{ precio | number:2 }}
```

Cabe decir que estas expresiones no están pensadas para escribir código de tu programa, osea, lo que llamamos lógica de tu aplicación. Como decíamos, están sobre todo pensadas para volcar información que tengas en tu modelo y facilitar el "binding". Por ello, salvo el operador ternario ($x ? y : z$), no se pueden colocar expresiones de control como bucles o condicionales. Sin embargo, desde las expresiones también podemos llamar a funciones codificadas en Javascript (que están escritas en los controladores, como veremos enseguida) para poder resolver cualquier tipo de necesidad más compleja.

Hasta aquí hemos descrito todos los componentes de AngularJS que ya conociste en el artículo de los [Primeros pasos con AngularJS](#). Ahora vamos a conocer otros componentes de las aplicaciones que podemos hacer con este framework Javascript.

Este artículo es obra de *Alberto Basalo*

Fue publicado por primera vez en 15/09/2014

Disponible online en <http://desarrolloweb.com/articulos/directivas-expresiones-angularjs.html>

Directivas ngInit, ngRepeat y ngClick

Ejemplo AngularJS en el que vamos a poder explorar el desarrollo únicamente con código HTML y aprender las directivas ngInit, ngRepeat y ngClick.

Seguimos aprendiendo AngularJS y en este artículo vamos a avanzar un poquito más y hacer un ejemplo práctico en el que no tenemos todavía necesidad de escribir código Javascript. Es interesante entretenerse con estos ejemplos, pues resultan muy sencillos y nos ayudan a mantener una progresión muy asequible en el principio de nuestro aprendizaje. También es apropiado para observar bien lo que a veces llamamos la "magia de AngularJS". Veremos que, con muy poco o nada de código, se pueden hacer cosas medianamente importantes, al menos para los desarrolladores que sabemos lo laborioso que sería montar esto por nuestra cuenta con Javascript a secas.

Nos servirá también para aprender nuevas directivas, de las más sencillas y usadas habitualmente, como son ngInit y ngClick o ngRepeat. Así que si os parece comenzamos con un poco de teoría para explicar estas directivas.



Directiva ngInit

Esta directiva nos sirve para inicializar datos en nuestra aplicación, por medio de expresiones que se evaluarán en el contexto actual donde hayan sido definidas. Dicho de otra manera, nos permite cargar cosas en nuestro modelo, al inicializarse la aplicación.

Así de manera general podemos crear variables en el "scope", inicializarlas con valores, etc. para que en el momento que las vayas a necesitar estén cargadas con los datos que necesitas.

Nota: Realmente no hemos explicado todavía en el [Manual de AngularJS](#), al menos suficientemente bien, qué es eso del "scope" y ya estamos abusando en el número de veces que usamos el término. Por ahora hemos comentado que el scope mantiene los datos de tu aplicación, lo que en el paradigma de la programación MVC se llama "modelo". Así pues, si digo que creo variables en el scope lo puedes entender como la definición de un dato de tu aplicación, que se guarda en el denominado "modelo" del MVC. Para los que tienen dudas sobre qué es el MVC os recomendamos leer el artículo [Qué es MVC](#).

```
<div ng-app ng-init="miArrayDatos = [];">
```

Con esto consigues que tu aplicación inicialice en el scope un dato llamado miArrayDatos como un array vacío. Pero no le prestes demasiada atención al hecho de haber colocado la directiva ngInit dentro de la misma etiqueta que inicializa la aplicación, pues podría ir en cualquier otra etiqueta de tu HTML. Realmente, colocarla en esa división marcada con ngApp es considerado una mala práctica. Ten en cuenta lo siguiente cuando trabajes con ngInit:

El único caso apropiado donde se debería de usar ngInit es en enlace de propiedades especiales de ngRepeat. Si lo que quieres es inicializar datos en tu modelo para toda la aplicación, el lugar apropiado sería en el controlador. Enseguida vemos un ejemplo de uso apropiado, cuando conozcamos la directiva ngRepeat.

Nota: Aclaremos ya que en el ejemplo final de este artículo vamos a usar ngInit para asociar datos al scope, e inicializarlos. Sin embargo, debemos de ser conscientes que para ello deberíamos usar un controlador. Como no hemos llegado a los controladores todavía, y queremos limitarnos solo a trabajar con el HTML, nos sirve como método alternativo.

Directiva ngRepeat

Esta directiva te sirve para implementar una repetición (un bucle). Es usada para repetir un grupo de etiquetas una serie de veces. Al implementar la directiva en tu HTML tienes que decirle sobre qué estructura se va a iterar. ngRepeat se usa de manera muy habitual y se verá con detenimiento en decenas de ejemplos. De momento puedes quedarte que es como un recorrido for-each en el que se itera sobre cada uno de los elementos de una colección.

La etiqueta donde has colocado el atributo ng-repeat y todo el grupo de etiquetas anidadas dentro de ésta, funciona como si fuera una plantilla. Al procesarse el compilador HTML de AngularJS el código HTML de esa plantilla se repite para cada elemento de la colección que se está iterando. Dentro de esa plantilla tienes un contexto particular, que es definido en la declaración de la directiva, que equivale al elemento actual en el bucle. Se ve mejor con un ejemplo.

```
<p ng-repeat="elemento in miColeccion">
  Estás en: <span>{{elemento}}</span>
</p>
```

El dato miColeccion sería un dato de tu modelo, habitualmente un array sobre el que puedas iterar, una vez por cada elemento. Pero también podría ser un objeto y en ese caso la iteración se realizaría en cada una de sus propiedades.

En lo relativo al contexto propio del bucle te puedes fijar que dentro de la iteración podemos acceder al dato "elemento", que contiene como valor, en cada repetición, el elemento actual de la colección sobre el que se está iterando.

Esta directiva es bastante sofisticada y explicar cada una de sus posibilidades nos llevaría algo de tiempo. Sin embargo, vamos a mostrar cómo podemos trabajar con ngRepeat en conjunto con ngInit, para completar la explicación de punto anterior.

```
<p ng-repeat="elemento in miColeccion" ng-init="paso=$index;">
  Elemento con id {{paso}}: <span>{{elemento}}</span>
</p>
```

La directiva ngRepeat maneja una serie de propiedades especiales que puedes inicializar para el contexto propio de cada repetición. Para inicializarlas usamos la directiva ngInit indicando los nombres de las variables donde vamos a guardar esas propiedades. En este caso estamos indicando que dentro de la repetición vamos a mantener una variable "paso" que tendrá el valor de \$index, que equivale al número índice de cada repetición. Osea, en la primera iteración paso valdrá cero, luego valdrá uno y así todo seguido. Igual que tienes \$index, Angular te proporciona otras propiedades útiles como \$first (que valdrá true en caso que sea la primera iteración) o \$last (true solo para la última iteración)

Nota: El que acabas de ver sería el único uso de ngInit considerado como buena práctica, en conjunto con ngRepeat para inicializar sus propiedades.

Dejamos de momento ngRepeat, aunque vuelvo a señalar que hay otra serie de cosas interesantes en la directiva para las repeticiones, como filtros, órdenes, etc.

Directiva ngClick

Terminamos nuestra dosis de teoría con una explicación de la directiva ngClick. Como podrás imaginarte es utilizada para especificar un evento click. En ella pondremos el código (mejor dicho la expresión) que se debe ejecutar cuando se produzca un clic sobre el elemento donde se ha colocado la directiva.

Típicamente al implementar un clic invocarás una función manejadora de evento, que escribirás de manera separada al código HTML.

```
<input type="button" value="Haz Clic" ng-click="procesarClic()">
```

Esa función procesarClic() la escribirás en el controlador, factoría, etc. Sería el modo aconsejado de proceder, aunque también podrías escribir expresiones simples, con un subconjunto del código que podrías escribir con el propio Javascript. Incluso cabe la posibilidad de escribir varias expresiones si las separas por punto y coma.

```
<input type="button" value="haz clic" ng-click="numero=2; otraCosa=dato " />
```

No difiere mucho a como se expresan los eventos clic en HTML mediante el atributo onclick, la diferencia aquí es que dentro de tus expresiones podrás acceder a los datos que tengas en tu modelo.

Nota: Aunque técnicamente pueda escribir expresiones directamente en el código HTML, en el valor del atributo ng-click, tienes que evaluar con cuidado qué tipo de código realizas porque dentro de la filosofía de AngularJS y la del MVC en general, no puedes escribir en tu HTML código que sirva para implementar la lógica de tu aplicación. (El código que necesitas para hacer las funcionalidades de tu aplicación no lo colocas en la vista, lo colocas en el controlador).

Ejemplo de uso de estas directivas en AngularJS

Ahora que hemos conocido las directivas, nos falta ponerlo todo junto para hacer un pequeño ejercicio básico con AngularJS.

Nota: Como advertimos al principio del artículo, solo voy a usar código HTML para este ejercicio. Por ese motivo hay un par de cosas que no serían consideradas buenas prácticas, como el uso de ngInit para inicializar el array "pensamientos" y el hecho de haber colocado dentro de un ng-click expresiones que sería mejor haber escrito en una función de nuestro controlador.

En esta aplicación tenemos un campo de texto para escribir cualquier cosa, un "pensamiento". Al pulsar sobre el botón se agregará dentro de un array llamado "pensamientos" lo que se haya escrito en el campo de

texto. Además encuentras un bucle definido con ng-repeat que itera sobre el array de "pensamientos" mostrando todos los que se hayan agregado.

```
<div ng-app ng-init="pensamientos = [];">
  <h1>Altavoz AngularJS</h1>
  <p>
    ¿Qué hay de nuevo?
    <br />
    <input type="text" ng-model="nuevoPensamiento" />
    <input type="button" value="Agregar" ng-click="pensamientos.push(nuevoPensamiento); nuevoPensamiento = '';" />
  </p>
  <h2>Pensamientos que has tenido</h2>
  <p ng-repeat="pensamiento in pensamientos" ng-init="paso = $index">
    Pensaste esto: {{pensamiento}} (Iteración con índice {{paso}})
  </p>
</div>
```

El dato del array "pensamientos" lo generas en el scope con el ng-init de la primera etiqueta. En el campo de texto tenemos la directiva ng-model para indicarle que lo que se escriba formará parte de nuestro modelo y se almacenará en la variable nuevoPensamiento. Como ves, en el ng-click se hace un push de ese nuevo pensamiento dentro del array de pensamientos. En ng-repeat se itera sobre la colección de pensamientos, escribiéndolos todos por pantalla, junto con el índice de ese pensamiento actual en el array "pensamientos".

Puedes ponerlo en marcha y jugar un poco con la aplicación haciendo tus cambios para calmar tu curiosidad o responder posibles dudas.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 16/09/2014
Disponible online en <http://desarrolloweb.com/articulos/directivas-nginit-ngrepeat-ngclick-angularjs.html>

Videotutoriales AngularJS, solo con directivas en HTML

Ejemplos prácticos de AngularJS en los que solo estamos realizando código HTML, con directivas embutidas en el código HTML.

En lo que hemos visto hasta aquí del [Manual de AngularJS](#) ya hemos tenido ocasión de conocer varias cosillas simples que nos sirven para hacer ejemplos. He hecho a lo largo de los artículos anteriores ya hemos podido hacer algunas muestras de cómo trabajar con el framework.

Queremos ahora introducir unos vídeos que nos van a aclarar todavía un poco más los primeros pasos en AngularJS. Estos vídeos están extractados del [#programadorIO de AngularJS desde cero punto cero](#). En este artículo además compartimos el código fuente utilizado.



1) "Hola mundo" en AngularJS

En este primer ejercicio vimos lo mínimo necesario para comenzar una aplicación con AngularJS.

Revisamos el uso de la directiva ngApp, la directiva ngModel y una expresión en la que volcamos un dato que hay en el modelo. Es muy sencillo, así que nadie se puede perder.

Para ver este vídeo es necesario visitar el artículo original en:
<http://desarrolloweb.com/articulos/videotutoriales-angularjs-directivas-html-ejemplos-practicos.html>

El código fuente realizado es el siguiente:

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
  <meta charset="UTF-8">
  <title>hola mundo en Angular</title>
</head>
<body>

  <input type="text" ng-model="campotexto">
  <br />
  {{ campotexto }}

  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular.min.js"></script>
</body>
</html>
```

2) Campos select y recomendaciones de inicialización

En este ejercicio incorporamos un campo SELECT que también asociamos al modelo de nuestra aplicación mediante la directiva ngModel. Gracias a ello podemos mostrar dentro del cuerpo de la página el valor del SELECT por medio de una expresión.

Para ver este vídeo es necesario visitar el artículo original en:

<http://desarrolloweb.com/articulos/videotutoriales-angularjs-directivas-html-ejemplos-practicos.html>

El código es el siguiente:

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
  <meta charset="UTF-8">
  <title>hola mundo en Angular</title>
</head>
<body>

  <input type="text" ng-model="campotexto">
  <br />
  {{ campotexto }} - {{ camposelect }}
  <br />
  <select ng-model="camposelect">
    <option value="uno">1</option>
    <option value="dos">2</option>
  </select>

  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular.min.js"></script>
</body>
</html>
```

3) Directiva ngClick

En este tercer vídeo vamos a explicar la directiva ngClick que ya nos permite agregar algo de interacción en los elementos de la página. Colocaremos un botón y dentro de él el atributo ng-click que nos permitirá asignarle un comportamiento.

Aclaremos en el vídeo que no es la manera más recomendable de actuar, puesto que los comportamientos deberían escribirse (su código) dentro de los controladores. No obstante nos ofrece una idea de las posibilidades de AngularJS y está bien para comenzar a dar los primeros pasos.

Mediante el botón escribimos dos comportamientos en el vídeo (aunque en el código fuente que tenemos solo se conservó uno de los comportamientos). El primero que verás en la grabación es asignarle un valor al campo SELECT. De ese modo se muestra cómo el campo de selección pierde el valor a null con el que lo inicializa automáticamente AngularJS. En el segundo comportamiento simplemente se coloca un nuevo texto en el campo de texto.

Para ver este vídeo es necesario visitar el artículo original en:

<http://desarrolloweb.com/articulos/videotutoriales-angularjs-directivas-html-ejemplos-practicos.html>

Este es el código fuente que quedó del ejemplo realizado en el vídeo.

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
  <meta charset="UTF-8">
  <title>hola mundo en Angular</title>
</head>
<body>

  <input type="text" ng-model="campotexto">
  <br />
  {{ campotexto }} - {{ camposelect }}
  <br />
  <select ng-model="camposelect">
    <option value="uno">1</option>
    <option value="dos">2</option>
  </select>
  <input type="button" ng-click="campotexto='DesarrolloWeb.com'" value="Pulsa aquí">

  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular.min.js"></script>
</body>
</html>
```

Estos son los tres ejemplos que te mostramos por ahora, aunque la sesión se completó con dos nuevas muestras de uso de AngularJS, en las que se mostró cómo crear un controlador. Sin embargo, como no hemos visto controladores todavía en este manual, nos reservamos para un poco más adelante comentarte acerca de esos vídeos. Si te interesan, lo único que debes hacer es irte a la [lista de reproducción de los videotutoriales de Angular en nuestro canal de Youtube](#).

En el siguiente artículo continuaremos hablando de modelos y luego de controladores, con lo que podremos comenzar a ver código fuente Javascript con buenas prácticas, en vez de mezclar la interacción con la presentación, tal como nos hemos obligado a hacer con lo que sabemos en estos primeros pasos con la librería Javascript.

De momento te sugiero que practiques por tu cuenta y trates de hacer ejemplos como éstos u otros más complejos que se te ocurran, pues con la práctica es como se afianzan los conocimientos. Observarás que jugar con AngularJS se hace muy divertido y que se consiguen comportamientos interesantes con muy poco o ningún código Javascript.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 22/09/2014
Disponible online en <http://desarrolloweb.com/articulos/videotutoriales-angularjs-directivas-html-ejemplos-practicos.html>

Separando el código Javascript

Después de aprender el desarrollo declarativo, conocemos las prácticas habituales en AngularJS, que nos indican que cada código debe ir en su lugar adecuado. Conocemos primeramente los módulos y controladores que nos permitirán organizar nuestro código separando el Javascript del HTML.

Módulo, objeto module en AngularJS

Qué son los módulos en AngularJS como crear un objeto module en el framework y qué tipo de cosas se podrán hacer con ellos.

Los módulos son una de las piezas fundamentales en el desarrollo con AngularJS y nos sirven para organizar el código en esta librería. Lo puedes entender como un contenedor donde situas el código de los controladores, directivas, etc.

La incorporación de módulos en AngularJS es motivada por la realización de aplicaciones con mejores prácticas. Son como contenedores aislados, para evitar que tu código interactúe con otros scripts Javascript que haya en tu aplicación (entre otras cosas dejarán de producirse colisiones de variables, nombres de funciones repetidos en otras partes del código, etc.). Los módulos también permiten que el código sea más fácilmente reutilizable, entre otras ventajas.



Indicar el módulo al arrancar la aplicación

Cuando arrancas una aplicación AngularJS, usando ng-app generalmente pondrás el nombre del módulo que quieres ejecutar en tu aplicación.

```
<HTML ng-app="miAplicacion">
```

Nota: La manera recomendada de arrancar la aplicación es con la directiva ngApp, aunque también podrías si lo deseas arrancarla con el método `angular.bootstrap()`.

```
angular.bootstrap(document.getElementById('miapp'), [modulos opcionales]);
```

En ese caso se supone que para identificar tu aplicación dentro del DOM le has colocado un identificador "miapp". Aprecia también que los módulos son opcionales e incluso que podría haber varios módulos que desees arrancar, algo posible, pero raramente usado.

Recuerda que hasta ahora, en lo que llevamos del [Manual de AngularJS](#), en la directiva ngApp, atributo ng-app, no colocábamos ningún valor. Eso era porque hasta el momento no habíamos trabajado con módulos. Sin embargo, a partir de ahora que los aprendemos a utilizar recuerda que los vas a tener que indicar al arrancar la aplicación.

Nota: Aunque en varios lugares de este manual nos podamos referir a ngApp como "directiva" queremos aclarar que realmente no lo es. [La propia documentación de AngularJS también la llama así](#), pero simplemente es un atributo y sirve para arrancar desde la función Bootstrap, que es llamada desde la función angularInit. <https://github.com/angular/angular.js/blob/master/src/Angular.js> Tampoco queremos liarte de momento con estos detalles, solo era una aclaración que pensamos es importante hacer.

Creación de módulos

Desde Javascript crearás los módulos usando el método angular.module() e indicándole una serie de parámetros. Echa un vistazo a esta sintaxis.

```
angular.module('miAplicacion', [ ... ], function(...){ ... })
```

La variable "angular" la tienes como variable global cuando cargas AngularJS, dentro tiene un objeto que estará disponible en cualquier parte de tu código. Luego ves "module", que es un método del objeto "angular" y que sirve para crear el módulo. El primer argumento es el nombre del módulo, que corresponde con el nombre de tu aplicación. En el segundo parámetro puedes indicar una serie de módulos adicionales, separados por comas, que serían tus dependencias. Pueden ser módulos de otros autores o puede que tú mismo hayas decidido organizar tu código en diferentes módulos. El tercer parámetro es opcional y en él colocamos una función que sirve para configurar AngularJS.

Para comenzar crearás tus módulos probablemente de una manera sencilla, tal como puedes ver a continuación:

```
angular.module('nuevaApp', []);
```

En el código anterior "nuevaApp" es el nombre de tu módulo. Como se comentó anteriormente, ese módulo se debe indicar en el bootstrap (arranque) de tu aplicación, comúnmente en la directiva ngApp.

```
<html ngApp="nuevaApp">
```


El objeto module

Esta llamada a `angular.module()` te devuelve un objeto `module`, que tiene una serie de métodos como `config`, `run`, `provider`, `service`, `factory`, `directive`, `controller`, `value`, etc. que son los que nos sirven para controlar la lógica de presentación y la lógica de negocio. Verás dos variantes de código en este sentido:

OPCIÓN 1) Puedes "cachear" (almacenar) el módulo en una variable y luego usarlo para crear tus controladores, factorías, etc.

```
var nuevaApp = angular.module('nuevaApp', []);
nuevaApp.controller( ... );
```

OPCIÓN 2) Aunque en muchos casos verás que ese objeto `module` ni siquiera se guarda en una variable, sino que se encadenan las creaciones de los controladores o factorías todo seguido. A esto último se suele llamar estilo "Fluent"

```
angular
  .module('nuevaApp', [])
  .controller( ... )
  .factory( ... );
```

Estas dos opciones proponen estilos de código distintos. En la segunda opción, al encadenar las llamadas desde la creación del módulo hasta los controladores u otras cosas que necesitemos, nos estamos ahorrando la creación de una variable global, eliminando posibilidad de que colisione con otras del mismo nombre que haya en otras partes de tu aplicación. Como te imaginarás, se recomienda la OPCIÓN 2).

Nota: Los módulos tienen mucha importancia en AngularJS, por lo que aprenderemos a hacer muchas cosas con ellos. De momento me interesan para crear controladores y poder avanzar un poco en los ejemplos prácticos del [Manual de AngularJS](#), así que no nos vamos a detener mucho en ellos todavía.

Acceder a un módulo ya creado en nuestra aplicación

El módulo lo crearás una vez, pero puedes necesitarlo en diversos puntos de tu código. Si lo has cacheado en una variable global, tal como se comentaba en la OPCIÓN 1) del punto anterior, podrás acceder a él en cualquier lugar de tu aplicación. Sin embargo, habíamos dicho que no era la opción más recomendable, así que necesitamos alguna otra manera de acceder a un módulo ya creado.

Lo conseguimos por medio del mismo método `module()` que utilizamos para crear el módulo, solo que en esta ocasión solo le indicaremos el módulo al que queremos acceder.

```
angular.module('nuevaApp')
```

Por si no has visto la diferencia con `angular.module('nombreModulo', [])` creamos un módulo y con `angular.module('nombreModulo')` recuperamos un módulo que haya sido creado anteriormente con el nombre 'nombreModulo'.

Módulos en adelante

Me gustaría terminar el artículo con un ejemplo práctico de uso módulos en AngularJS pero si utilizamos simplemente el nuevo conocimiento de este artículo sería complicado. En realidad los módulos sirven como base para implementar muchas cosas en AngularJS, por lo que enseguida les sacaremos bastante provecho.

En el siguiente artículo nos introduciremos en los controladores y comenzaremos a utilizar el famoso "scope". Para todo ello es condición indispensable comenzar con un módulo, así que las prácticas en esta ocasión las dejaremos para más adelante.

Este artículo es obra de *Pedro Hurtado*
Fue publicado por primera vez en 29/09/2014
Disponible online en <http://desarrolloweb.com/articulos/modulo-objeto-module-angularjs.html>

Controladores, controller en AngularJS

Comenzamos a entender los controladores. Conocemos lo que es un controller en AngularJS, para qué se usa y qué tipo de código deben tener, introducimos también el concepto de scope en Angular.

Los controladores nos permiten mediante programación implementar la lógica de la presentación en AngularJS. En ellos podemos mantener el código necesario para inicializar una aplicación, gestionar los eventos, etc. Podemos decir que gestionan el flujo de la parte del cliente, lo que sería programación para implementar la funcionalidad asociada a la presentación.

En líneas generales podemos entender que los controladores nos sirven para separar ciertas partes del código de una aplicación y evitar que escribamos Javascript en la vista. Es decir para que el HTML utilizado para la presentación no se mezcle con el Javascript para darle vida.

Nota: Los que ya conocen el MVC podrán hacerse una idea más concreta del cometido de los controladores. Si no es tu caso, no te preocupes porque en futuros artículos podrás leer de manera teórica el MVC y de manera particular cómo se aplica este paradigma en AngularJS. Para los que no sepan MVC y quieran aprender de qué se trata en general, recomendamos la lectura del artículo sobre el [paradigma MVC](#).



Un controlador puede ser agregado al DOM mediante la directiva `ngController` (con el atributo `ng-controller` en la etiqueta HTML) y a partir de entonces tendremos disponible en esa etiqueta (y todas sus hijas) una serie de datos. Esos datos son los que necesitarás en la vista para hacer la parte de presentación y es lo que asociaríamos en el MVC con el "modelo". En la terminología de Angular al modelo se le llama "scope" y dentro de poco vamos a explicarlo un poco mejor.

A los controladores podemos inyectarles valores o constantes. Como su propio nombre indica, las constantes son las que no van a cambiar a lo largo del uso de la aplicación y los valores son aquellas variables cuyo dato puede cambiar durante la ejecución de una aplicación. También podremos inyectar servicios y factorías, componentes muy parecidos entre sí y que veremos más adelante.

Nota: Sin entrar a describir de momento lo que son "services" y "factories" puedes simplemente recordar que su utilidad es la de encapsular código.

Concepto de Scope

Para poder introducir los controladores debemos detenernos antes en un concepto que se repite hasta la saciedad dentro de la literatura de AngularJS, el "scope". De hecho, tal como dice la documentación de AngularJS, el cometido de un controlador consiste en desempeñar una función constructora capaz de aumentar el Scope.

El "scope" es la pieza más importante del motor de AngularJS y es donde están los datos que se tienen que manejar dentro de la parte de presentación.

El scope es un gran contenedor de datos, que transporta y hace visible la información necesaria para implementar la aplicación, desde el controlador a la vista y desde la vista al controlador. En términos de código el scope no es más que un objeto al que puedes asignar propiedades nuevas, con los datos que necesites, o incluso con funciones (métodos).

Esos datos y esas funciones están visibles tanto en el Javascript de los controladores como en el HTML de las vistas, sin que tengamos que realizar ningún código adicional, pues Angular ya se encarga de ello automáticamente. Además, cuando surgen cambios en los datos se propagan entre los controladores y las vistas automáticamente. Esto se realiza por un mecanismo que llamamos "binding", y en AngularJS también "doble binding" (en español sería enlace), que explicaremos con detalle en futuros artículos.

Así pues, desde los controladores vamos a ser capaces de trabajar con el scope de una manera minuciosa,

agregando o modificando información según lo requiera nuestra aplicación.

Nota: Hasta el momento en este [Manual de AngularJS](#) hemos usado el scope en varias ocasiones y ya hemos dicho que representa al modelo, del paradigma MVC. Siempre que hemos usado el scope en HTML ha sido 1) agregando partes de la página al modelo mediante la directiva ngModel, por ejemplo datos que se escriban en un campo de texto o se seleccionen en un campo select, o 2) volcando información en el HTML mediante expresiones escritas entre dos llaves {{ }}.

Todo eso estaba muy bien, sin embargo en términos de programación necesitamos un lugar donde escribir todo el Javascript necesario para implementar la lógica de la aplicación. Integrar el Javascript dentro del HTML no es nada recomendable, por diversos motivos que ya se conocen. Ya dijimos además que dentro del código HTML, no se puede (o mejor dicho, no se debería) hacer cálculos, asignaciones de valores y en resumen código que represente la lógica de nuestras aplicaciones. Todo ello irá en los controladores.

Qué hacer y qué no hacer desde los controladores

Ahora vamos a ver qué tipo de operaciones debemos incluir dentro del código de los controladores. Entre otras cosas y por ahora debes saber:

- **Los controladores son adecuados para inicializar el estado del scope** para que nuestra aplicación tenga los datos necesarios para comenzar a funcionar y pueda presentar información correcta al usuario en la vista.
- Además es **el lugar adecuado para escribir código que añada funcionalidades o comportamientos** (métodos, funciones) al scope.

Con el controlador **no deberías en ningún caso manipular el DOM de la página**, pues los controladores deben de ser agnósticos a cómo está construido el HTML del documento donde van a estar trabajando.

Tampoco son adecuados para formatear la entrada de datos o filtrar la salida, ni intercambiar estados entre distintos controladores. Para hacer todo eso existen dentro de AngularJS diversos componentes especializados.

Cómo es un controlador en AngularJS

El código necesario para crear un controlador en AngularJS tendrá este aspecto:

```
var app = angular.module("miapp", []);
app.controller("miappCtrl", function(){
    var scope = this;
    scope.datoScope = "valor";

    scope.metodoScope = function(){
        scope.datoScope = "otra cosa";
    }
});
```

Como puedes comprobar, resulta un tanto complejo. Además tenemos que explicarte cómo se engancha este controlador dentro del código HTML. Para explicar todo eso con calma necesitamos alargarnos bastante, así que lo veremos en el siguiente artículo. Además, te explicaremos varias alternativas de código para trabajar con controladores que se usan habitualmente en AngularJS, con mejores y peores prácticas.

Este artículo es obra de *Xavier Jorge Cerdá*
Fue publicado por primera vez en 06/10/2014
Disponible online en <http://desarrolloweb.com/articulos/controladores-controller-angularjs.html>

Variantes para crear controladores en AngularJS

Este artículo contiene varias aproximaciones a controladores creados en AngularJS con diferentes estilos de codificación que nos aportan diferentes posibilidades.

De momento está bien de teoría. Después de aprender [qué son los controladores y qué funciones desempeñan en Angular](#), estamos seguros que querrás poner manos en el código y lo vamos a hacer con un nuevo ejercicio que, aun siendo sencillo, nos facilite experimentar con un primer controlador.

Además, estudiaremos diversas variantes que nos pueden aportar algunas ventajas, inconvenientes y resolver ciertas situaciones. Las veremos todas con ejemplos y las explicaciones sobre qué diferencias nos encontramos.

El proceso de crear y usar un controlador es sencillo, pero no inmediato, requiere un pequeño guión de tareas que paso a detallar para que tengas claro de antemano lo que vamos a hacer. Es más largo expresarlo con texto que con código y aunque al principio parezca bastante, en seguida lo harás mecánicamente y no tendrás que pensar mucho para ello. Básicamente para poner nuestro primer controlador en marcha necesitamos:

- Crear un módulo (module de AngularJS)
- Mediante el método `controller()` de nuestro "module", asignarle una función constructora que Angular usará cuando deba crear nuestro controlador
- Usar la directiva `ng-controller`, asignándole el nombre de nuestro controlador, en el HTML. Colocaremos esa directiva en el pedazo del DOM donde queremos tener acceso al scope.



En este punto queremos aclarar que existen diversas alternativas de código para crear el Javascript necesario para definir nuestro controlador. Vamos a comenzar por una que es muy habitual de ver en diversos tutoriales y documentación, aunque no es la mejor. Todas las formas funcionan, pero hay algunas que en determinadas circunstancias pueden ser mejores. Luego las estudiaremos y explicaremos el motivo por el que serían todavía más recomendadas para crear tus controladores.

Nota: El objetivo de este primer controlador es simplemente observar cuál es el código y la sintaxis para crearlos. Realmente observarás es un controlador muy tonto, pero espero me disculpes. Buscaremos más adelante ejemplos más útiles.

OPCIÓN 1)

```
angular
  .module('pruebaApp', [])
  .controller('pruebaAppCtrl', function($scope){
    $scope.algo = "Hola Angular, usando controller más simple";
  });
```

De esta manera estoy creando un módulo llamado "pruebaApp". Sobre el módulo invoco el método `controller()` y creo un controlador llamado "pruebaAppCtrl". En la creación del controlador verás que se especifica una función, ella es la que hace de constructora de nuestro controlador. A esa función le llega como parámetro `$scope` que es el nombre de variable donde tendrás una referencia al conodido "scope" que almacena los datos de mi modelo.

Dentro de la función del controlador hacemos poca cosa. De momento simplemente le asigno un valor a un atributo de `$scope`. He creado un atributo nuevo en el objeto `$scope` que no existía y que se llama "algo" (`$scope.algo`). Con ello simplemente le estoy agregando un dato al modelo. Gracias al "binding" o enlace, ese dato luego lo podré utilizar en el HTML.

De hecho, queremos ya echarle un vistazo al HTML que podríamos tener para usar este controller.

```
<div ng-app="pruebaApp" ng-controller="pruebaAppCtrl">
  {{algo}}
</div>
```

Como puedes ver, en el HTML indicamos el nombre del módulo que se usará para implementar esta aplicación (que corresponde con el nombre del módulo creado en el código Javascript) y el nombre del controlador (también corresponde con el nombre del controller que he creado con Javascript). Además en el HTML podrás encontrar la expresión `{{algo}}` que lo que hace es volcar como texto en la página el contenido de la variable "algo". Esa variable pertenece al scope y la hemos inicializado en el código del controller.

El resultado es que en nuestra página aparecerá el texto "Hola Angular, usando controller", que es el valor que fue asignado en el atributo "algo" de `$scope`. De momento la aplicación no es nada espectacular pero lo

interesante es ver cómo se usa un controller.

VARIANTE OPCIÓN 1)

Existe una variante de este mismo ejemplo, al crear el controlador, que deja nuestro código un poco más complejo, pero que ayuda en determinadas circunstancias. La clave aquí son los minimificadores de Javascript (o minificadores), que sabemos que reducen el peso en bytes del código compactándolo de distintas maneras. Si los usas, el código que hemos visto en el anterior apartado puede dar errores una vez minimificado.

El problema que puede surgir está en la inyección del `$scope`, que se recibe como parámetro en la función constructora del controller.

```
function($scope)
```

Angular, al ver el nombre del parámetro `"$scope"`, sabe perfectamente lo que tiene que inyectar en ese lugar. Sin embargo, una vez minimificado, el nombre de esa variable `"$scope"` puede haberse cambiado por algo como `"a"`, lo que haría imposible que el framework sepa lo que debe de enviar. Eso se soluciona con un código como el siguiente.

```
angular
  .module('pruebaApp', [])
  .controller('pruebaAppCtrl', ['$scope', function($scope){
    $scope.algo = "Hola Angular, usando controller!";
  }]);
```

En este caso, en la llamada al método que crea el controlador, `controller()`, ves un array que nos sirve para inyectar las dependencias.

```
['$scope', function($scope){ ... }]
```

En este caso le estamos diciendo que el primer parámetro del controlador es `'$scope'`, gracias al primer elemento del array. Como las cadenas como `'$scope'` no se alteran después de la minimización, podemos estar seguros que le llegarán perfectamente a Angular. Entonces sabrá que en el primer parámetro de la función constructora del controlador esperas recibir el `$scope`. Esto será así independientemente del nombre de la variable que se uses como primer parámetro.

Es cierto que la sintaxis parece un poco rebuscada, pero poco a poco te acostumbrarás, pues realmente es muy repetitivo.

Nota: Esto que acabamos de explicar para el tema de la inyección de dependencias sirve igual con muchos otros elementos que podrías necesitar en tus controladores. Osea, puedes inyectar del mismo modo otras cosas aparte del `$scope`, como `$http`, etc. Esta variante de la opción 1, debido al modo de declarar la inyección de dependencias, es más adecuada y es algo retomaremos también en futuras entregas de este manual.

OPCIÓN 2)

He colocado como primera opción porque es más habitual en tutoriales y puedes haberla leído en la documentación del framework o en diversas presentaciones por ahí. Sin embargo, es más recomendada esta segunda alternativa que os presentamos a continuación.

```
angular
  .module('pruebaApp', [])
  .controller("pruebaAppCtrl", function(){
    this.algo = "Esto funciona! Gracias Angular";
  });
```

Como estarás observando, el controlador es prácticamente igual. Ahora la diferencia es que no estás inyectando el \$scope. Ahora el contexto donde adjuntar las variables que queremos enviar a la vista no lo sacamos de \$scope, sino que lo obtenemos directamente a través de la variable "this".

En este caso, para cargar un nuevo dato al modelo, llamado "algo", lo hacemos a través de this.algo y le asignamos aquello que deseemos.

Esta alternativa implica algún cambio en el HTML con respecto a lo que vimos anteriormente.

```
<div ng-app="pruebaApp" ng-controller="pruebaAppCtrl as vm">
  {{vm.algo}}
</div>
```

El cambio fundamental lo encuentras al declarar la directiva, en el valor del atributo ng-controller.

```
ng-controller="pruebaAppCtrl as vm"
```

En este caso estamos diciéndole no solo el nombre del controlador, sino además estamos informando que dentro de ese DOM el scope será conocido con la variable "vm". Por tanto, ahora cuando deseemos acceder a datos de ese modelo que nos ofrece el controlador debemos indicar el nombre del scope.

```
{{vm.algo}}
```

Nota: La recomendación de esta opción 2 se basa en que así puedes tener varios controladores en tu página, incluso anidados unos dentro de otros, y los datos que nos ofrece cada uno están separados en distintos espacios de nombres. Esto quizás ahora no te dice nada, pero sí que evita en el futuro habituales problemáticas de desarrollo con AngularJS.

Ahora que hemos visto estas dos opciones y dado que nos recomiendan la segunda, la utilizaremos de manera preferente a lo largo del manual. De momento esto es todo, esperamos que con lo que sabes ya tengas ideas para ir probando nuevos ejemplos. Nosotros en el siguiente artículo crearemos un controlador que será un poco más complejo y útil.

Acabamos este artículo con un vídeo de nuestro canal de Youtube donde encontrarás más información sobre la creación de controladores en AngularJS. Veremos diferentes variantes del código realizadas en directo. También encontrarás explicaciones ofrecidas por varios compañeros expertos en este framework Javascript.

Para ver este vídeo es necesario visitar el artículo original en:

<http://desarrolloweb.com/articulos/ejercicio-controladores-angularjs.html>

Este artículo es obra de *Alberto Basalo*

Fue publicado por primera vez en 08/10/2014

Disponible online en <http://desarrolloweb.com/articulos/ejercicio-controladores-angularjs.html>

Segundo ejercicio con controller en AngularJS

Crearemos una aplicación sencilla con la intención de practicar con controller en AngularJS, inicializando datos en el Scope, creando manejadores de eventos, etc.

Vamos a hacer un ejercicio práctico de todo lo que venimos aprendiendo con AngularJS y donde afianzar particularmente lo aprendido sobre la [creación de controladores](#).

El ejercicio es muy sencillo en realidad. Es un "acumulador" en el que tenemos un campo de texto para escribir una cantidad y un par de botones con operaciones de sumar y restar. Luego tenemos un contador que se va incrementando con esas operaciones de sumas y restas sobre un total. En si no sirve para mucho lo que vamos a construir, lo hacemos más bien con fines didácticos.



Puedes ver el ejercicio en marcha en [Codepen](#) antes de pasar a estudiar y explicar el código con el que se ha hecho.

Código HTML

Veamos primero la parte del HTML usada para resolver este ejercicio.

```
<div ng-app="acumuladorApp" ng-controller="acumuladorAppCtrl as vm">
  <h1>Acumulador</h1>
  <h2>Control de operación:</h2>
  ¿Cuánto? <input type="text" ng-model="vm.cuanto" size="4" />
  <br />
  <input type="button" value="+" ng-click="vm.sumar()"/>
  <input type="button" value="-" ng-click="vm.restar()"/>
  <h2>Totales:</h2>
  En el acumulador llevamos <span>{{vm.total}}</span>
</div>
```

Voy describiendo los detalles más importantes que debes apreciar.

- Al hacer el bootstrap (arranque) de la aplicación (directiva ngApp) se indica el nombre del módulo: "acumuladorApp".
- Se indica el nombre del controlador con la directiva ngController y el valor "acumuladorAppCtrl".
- Con la sintaxis de "acumuladorAppCtrl as vm" indicamos que el scope dentro del espacio de etiquetas marcado para este controlador, se conocerá por "vm". Podríamos llamar como deseásemos al scope, en lugar "vm", en definitiva es como una variable donde tendremos propiedades y métodos. En otras palabras, se ha creado un namespace (espacio de nombres) para los datos que nos vienen del modelo (scope) gracias al controlador "acumuladorAppCtrl".
- Puedes llamar de cualquier manera también tanto a módulo como a controlador, pero se usan esos por convención.
- El campo INPUT de texto tiene una directiva ngModel para decirle a AngularJS que ese es un dato del modelo. Fíjate que el nombre del dato en el modelo se accede a través del espacio de nombres definido en el controlador: "vm.cuanto".
- Luego encuentras dos INPUT tipo button que me sirven para realizar la acción de acumular, positiva o negativamente. Ambos tienen una directiva ngClick que nos sirve para expresar lo que debe ocurrir con un clic. Lo interesante aquí es que llamamos a dos funciones que están definidas en el scope, mediante el espacio de nombres "vm", nuestro modelo. El código de esos métodos (funciones que hacen en este caso de manejadores de eventos) ha sido definido en el controlador, lo veremos más adelante.
- Por último encontramos un {{vm.total}} que es un dato que estará en el scope y en el que llevamos la cuenta de todo lo que se ha sumado o restado con el acumulador.

Nota: Por si no te has dado cuenta, el espacio de nombres donde tendremos al scope se ha nombrado como "vm". Ya dijimos que se puede usar cualquier nombre de variable, eso es lo de menos, lo que queremos que se note es que "vm" son las siglas de "View Model", osea, el "modelo de la vista". Es algo que tiene que ver con el paradigma MVC en la variante en la que trabajamos con Angular. Hablaremos de eso más adelante.

Código Javascript

Ahora pasemos a la parte donde codificamos nuestro Javascript para darle vida a este ejercicio.

```
angular
  .module('acumuladorApp', [])
  .controller("acumuladorAppCtrl", controladorPrincipal);

function controladorPrincipal(){
  //esta función es mi controlador
  var scope = this;
  scope.total = 0;
  scope.cuanto = 0;

  scope.sumar = function(){
    scope.total += parseInt(scope.cuanto);
  }
  scope.restar = function(){
    scope.total -= parseInt(scope.cuanto);
  }
};
```

Vamos describiendo las principales partes del código.

- Con "angular" en la primera línea accedo a la variable que me crea angular, disponible para acceder a las funcionalidades del framework.
- En la segunda línea creamos el módulo, indicando el nombre (que es igual a lo que se puso como valor en la directiva ngApp del HTML) y el array de las dependencias, de momento vacío.
- En la tercera línea, con un encadenamiento (chaining) definimos el controlador. Indicamos como primer parámetro el nombre del controlador, definido en la directiva ngController del HTML, y como segundo parámetro colocamos la función que se encargará de construir el controlador. Indicamos el nombre de la función simplemente, sin los paréntesis, pues no es una llamada a la función sino simplemente su nombre.
- Luego se define la función del controlador. Esa función es capaz de escribir datos en el scope, así como métodos.
- En la función accedemos al scope por medio de "this". Fíjate que en la primera línea de la función tienes var scope = this; esto es simplemente opcional y se puede hacer para mantener la terminología de AngularJS de llamar scope a lo que vas generando en el controlador, pero podría perfectamente referirme a él todo el tiempo con "this".
- En el scope inicializo dos valores, total y cuanto, mediante scope.total=0 y scope.cuanto=0.
- Luego genero dos métodos que usaremos para los manejadores de eventos de los botones de sumar y restar. Esas funciones tienen el scope disponible también y en su código se accede y modifica a los datos del scope.

Nota: También hay varias aproximaciones para definir este Javascript con los módulos y controladores de Angular. Es recomendable la creación de una envoltura para tu Javascript, por medio de una función que se autoinvoque. Esto lo veremos más adelante, puesto que no queremos complicar más el código todavía.

Con eso tenemos el ejercicio completo, todo lo demás para que esto funcione es tarea de AngularJS. De manera declarativa en el HTML hemos dicho qué son las cosas con las que se va a trabajar en la aplicación y luego hemos terminado de definir e inicializar los datos en el controlador, así como escribir en código las funcionalidades necesarias para que el ejercicio tome vida.

Variante de este mismo ejercicio pero sin "controller as"

En la resolución, que hemos comentado antes, a este ejercicio hemos usado una alternativa de la directiva de `ngController` en la que se le asigna un espacio de nombres al scope "acumuladorAppCtrl as vm". Esto se conoce habitualmente como "controller as" y ya comentamos en el artículo anterior dedicado a los controladores que es algo relativamente nuevo y que muchas veces la codificación que encontrarás en otros textos es un poco diferente.

Solo a modo de guía para quien está acostumbrado a trabajar de otra manera, y para que entiendas otros códigos antiguos que podrás encontrar en otras guías de Angular, pongo aquí el código de este mismo ejercicio pero sin el "controller as".

El código HTML

```
<div ng-app="acumuladorApp" ng-controller="acumuladorAppCtrl">
  <h1>Acumulador</h1>
  <h2>Control de operación:</h2>
  ¿Cuánto? <input type="text" ng-model="cuanto" size="4" />
  <br />
  <input type="button" value="+" ng-click="sumar()" />
  <input type="button" value="-" ng-click="restar()" />
  <h2>Totales:</h2>
  En el acumulador llevamos <span>{{total}}</span>
</div>
```

El código Javascript

```
var acumuladorApp = angular.module('acumuladorApp', []);
acumuladorApp.controller("acumuladorAppCtrl", ['$scope', function($scope){
  //esta función es mi controlador
  //var $scope = this;
  $scope.total = 0;
  $scope.cuanto = 0;

  $scope.sumar = function(){
```



```
$scope.total += parseInt($scope.cuanto);  
}  
$scope.restar = function(){  
    $scope.total -= parseInt($scope.cuanto);  
}  
});
```

No comento el código más, pues es muy parecido al anterior, simplemente se deja de usar el espacio de nombres y al definir la función del controlador se inyecta el \$scope de otra manera. Te dejo a ti la tarea de encontrar todas las diferencias.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 07/10/2014
Disponible online en <http://desarrolloweb.com/articulos/segundo-ejercicio-controller-angularjs.html>

Explorando directivas a fondo

Para trabajar con AngularJS necesitamos dominar una serie de directivas que se encuentran presentes en el núcleo del framework, además de apoyarnos en otras directivas creadas por la comunidad. Las directivas enriquecen nuestro HTML y nos permiten realizar operaciones útiles sobre la página. Ahora exploraremos algunas directivas fundamentales con mayor detalle.

Directiva ngClass en AngularJS

Explicaciones de las alternativas posibles de uso de la directiva ngClass de AngularJS con ejemplos prácticos.

Seguimos aprendiendo aspectos básicos sobre AngularJS y nos vamos a detener en una directiva muy útil que nos ayuda a definir clases CSS (class) que se le deben aplicar a los elementos bajo ciertas circunstancias.

La idea es poder definir el aspecto de nuestra aplicación en base a los datos que tengamos en el modelo, aplicando unas class de CSS u otras dependiendo de valores de propiedades del scope o de expresiones que podamos construir.



Existen tres posibles variantes de tipos que acepta la directiva ngClass y que podemos usar siempre que deseemos.

- Asignarle una propiedad del scope que sea una cadena de texto. En este caso esa cadena se coloca como valor en el atributo class. Si en esa cadena existen varios nombres de clases separados por un espacio en blanco, esas clases se aplicarán en conjunto al elemento.
- Asignarle una propiedad del scope que contenga un array de cadenas. En ese caso se asignan como clases todos los nombres que haya en las casillas del array.
- Asignarle como valor a ng-class un objeto. En ese caso tendrá pares clave valor para especificar nombres de clases y expresiones que deban cumplirse para que éstas se apliquen. Lo veremos mejor con un ejemplo.

Ahora veremos ejemplos de cada una de las tres posibilidades comentadas.

Ejemplo 1) Asignar una propiedad del scope que contiene una cadena

Es tan sencillo como indicar esa propiedad dentro del atributo ng-class, como se ve a continuación.

```
<h1 ng-class="vm.tamTitular">Acumulador</h1>
<select ng-model="vm.tamTitular">
  <option value="titularpeq">Peque</option>
  <option value="titulargran">Gran</option>
</select>
```

Como puedes ver, tenemos un encabezado H1 con ng-class asignado a vm.tamTitular. Esa propiedad del scope se creó a partir de un campo SELECT que está a continuación. Por tanto, cuando cambie el option seleccionado en el campo, cambiará la class asociada al elemento H1.

Tal como habrás deducido, las dos posibles class que se le van a asignar al encabezado serán los value de los OPTION.

Ejemplo 2) Asignar un array de cadenas

Es tan sencillo como definir un array de alguna manera y luego introducirlo dentro de nuestra directiva. Por facilitar las cosas voy a definir el array de manera literal en el controlador.

```
vm.clases = ["uno", "dos", "tres"];
```

Luego podremos asociar ese array de cadenas, colocando todos los nombres de clases a un elemento de la página con la directiva ngClass.

```
<h2 ng-class="vm.clases">Control de operación:</h2>
```

Como resultado de esa directiva a nuestro encabezado H2 se le van a aplicar las tres clases "uno", "dos" y "tres".

Ejemplo 3) Asignar un objeto con uno o varios pares clave, valor

Este es el uso más complejo de ngClass, pero también el más potente. Nos permite definir expresiones y gracias a ellas Angular sabrá si debe colocar o no una clase CSS en concreto. Se ve bien con un ejemplo delante.

```
<p ng-class="{positivo: vm.total>=0, negativo: vm.total<0}">
  En el acumulador llevamos <span>{{vm.total}}</span>
</p>
```

Este párrafo nos muestra un valor total de una cuenta. Ese valor lo sacamos de la propiedad vm.total y esa misma propiedad es la que usamos para definir la clase de CSS que vamos a asociar como estilo al párrafo.

Ahora echa un vistazo al atributo `ng-class` y verás que lo que le indicamos es un objeto, pues está entre llaves. Ese objeto tiene un número de pares clave/valor indeterminado, puede ser uno o varios. Cada uno de esos pares clave/valor nos sirven para definir si debe o no aplicarse una clase en concreto.

En la clave colocas el nombre de la clase, `class` de tu CSS, que Angular puede colocar si se cumple una expresión booleana. Como valor colocamos la expresión booleana a evaluar por AngularJS para que el sistema deduzca si se debe aplicar esa clase o no.

En nuestro ejemplo se aplicará la clase "positivo" en caso que la propiedad `vm.total` sea mayor o igual que cero. Se aplicará la clase "negativo" en caso que la propiedad `vm.total` tenga un valor menor que cero.

Ejercicio englobando estos tres usos de `ngClass`

Bien, con esto ya sabes todo lo que necesitas para poder trabajar con tus `ngClass`. A continuación encuentras un ejercicio en el que simplemente se usan estos mismos ejemplos relatados hasta ahora. Si lo has entendido no tendrás problema alguno al ver el código siguiente.

Nota: Para la realización de este código hemos partido como base del ejemplo relatado en el artículo [Segundo ejercicio con controller en AngularJS](#). En realidad solo hemos agregado los tres posibles usos de la directiva `ngClass` explicados en este artículo.

Lo vemos por partes. Primero el código HTML:

```
<div ng-app="acumuladorApp" ng-controller="acumuladorAppCtrl as vm">
  <h1 ng-class="vm.tamTitular">Acumulador</h1>
  <select ng-model="vm.tamTitular">
    <option value="titularpeq">Peque</option>
    <option value="titulargran">Gran</option>
  </select>
  <h2 ng-class="vm.clases">Control de operación:</h2>
  ¿Cuánto? <input type="text" ng-model="vm.cuanto" size="4" placeholder="0" />
  <br />
  <input type="button" value="+" ng-click="vm.sumar()" />
  <input type="button" value="-" ng-click="vm.restar()" />
  <h2>Totales:</h2>
  <p ng-class="{positivo: vm.total>=0, negativo: vm.total<0}">
    En el acumulador llevamos <span>{{vm.total}}</span>
  </p>
</div>
```

Ahora puedes ver el código CSS con la definición de varias clases con las que vamos a jugar.

```
.negativo {
  color: red;
}

.positivo {
  color: #33f;
```

```
}  
.titularpeq{  
  font-size: 10pt;  
}  
.titulargran{  
  font-size: 18pt;  
}  
.uno{  
  background-color: #666;  
}  
.dos{  
  color: #fff;  
}  
.tres{  
  font-size: 30px;  
}
```

Por último el código Javascript de nuestro controlador. Como decía, encontrarás diversas cosas adicionales a las comentadas en este artículo, pero no te deben extrañar porque fueron explicadas anteriormente en el [Manual de AngularJS](#).

```
angular  
  .module('acumuladorApp', [])  
  .controller("acumuladorAppCtrl", controladorPrincipal);  
  
function controladorPrincipal(){  
  //esta función es mi controlador  
  var vm = this;  
  vm.total = 0;  
  //scope.cuanto = 0;  
  vm.tamTitular = "titularpeq"  
  
  vm.sumar = function(){  
    vm.total += parseInt(vm.cuanto);  
  }  
  vm.restar = function(){  
    vm.total -= parseInt(vm.cuanto);  
  }  
  vm.clases = ["uno", "dos", "tres"];  
};
```

Eso es todo, esperamos que puedas disfrutar de esta útil directiva en tus proyectos. Con un poco de imaginación comprobarás la potencia de ngClass.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 27/10/2014
Disponible online en <http://desarrolloweb.com/articulos/directiva-ngclass-angularjs.html>

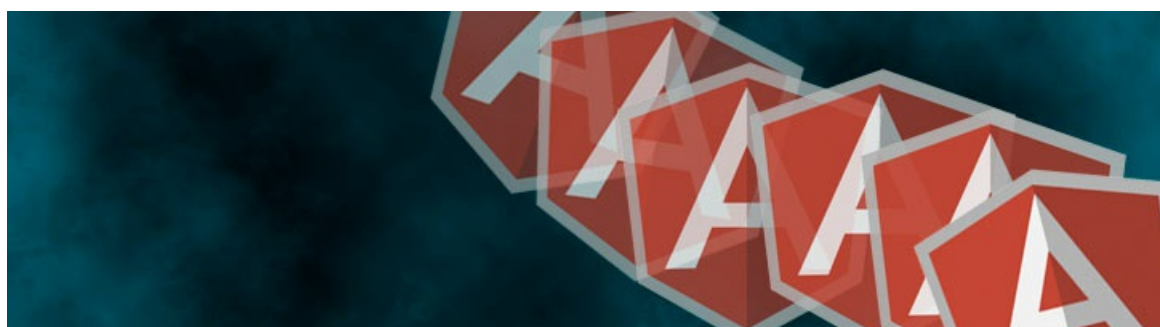
Trabajando con campos checkbox en AngularJS

Explicaciones y prácticas con campos input checkbox con AngularJS, conociendo las directivas `ngChecked`, `ngTrueValue`, `ngFalseValue`, `ngChecked`.

En el estilo de aplicaciones que se hacen con AngularJS trabajas de manera intensiva con campos de formulario. Puedes leer este artículo para aprender todo sobre los checkbox y hacer algún ejemplo práctico.

En Angular los campos input checkbox tienen una serie de directivas que podemos usar:

- **ngModel:** indica el nombre con el que se conocerá a este elemento en el modelo/scope.
- **ngTrueValue:** La utilizas si deseas asignar un valor personalizado al elemento cuando el campo checkbox está marcado.
- **ngFalseValue:** es lo mismo que `ngTrueValue`, pero en este caso con el valor asignado cuando el campo no está "chechado".
- **ngChange:** sirve para indicar expresiones a realizar cuando se produce un evento de cambio en el elemento. Se dispara cuando cambia el estado del campo, marcado a no marcado y viceversa. Podemos ejecutar nuestra expresión o llamar a una función en nuestro scope.



Nota: Además, tenemos `ngChecked`, aunque esta directiva no está en la documentación del campo checkbox, sino en otra página. <https://docs.angularjs.org/api/ng/directive/ngChecked> Viene a explicar que sirve para conservar un valor en el atributo `checked`. Ese atributo en HTML funciona como un booleano, si se encuentra el atributo se entiende que está chequeado y si no se encuentra `checked` es que no debe estar marcado. Por ello los navegadores no están obligados a memorizar ningún valor en el `checked`. En esos casos puedes usar `ngChecked`. En definitiva, en la práctica ese campo sirve para indicar una variable del scope donde se pueda deducir si el elemento debe estar o no marcado (`checked`).

Directiva ngModel

Si quieres usar un checkbox lo más normal es que indiques la referencia de tu modelo donde quieres que se guarde su estado.

```
<input type="checkbox" ng-model="vm.activo" />
```


A partir de este momento el checkbox está asociado a tu scope en `vm.activo`. Pero un detalle, en el scope todavía no está creada esa propiedad hasta que no pulses encima del checkbox para activarlo o desactivarlo. En ese momento pasa a existir `vm.activo` en el modelo, aunque también si lo deseamos podemos inicializarla en un controlador.

```
vm.activo = true;
```

Como sabes, durante la vida de tu aplicación, el estado del checkbox se traslada automáticamente desde la vista al modelo y desde el modelo a la vista, por el proceso conocido por "doble binding". En resumen, si en cualquier momento desde el Javascript cambias el valor de `vm.activo`, siempre se actualizará la vista. Por supuesto, si en la vista pulsas sobre el campo para activarlo o desactivarlo, en el modelo también quedará reflejado el nuevo estado.

Directivas `ngTrueValue` y `ngFalseValue`

En tu modelo, la propiedad `vm.activo` podrá tener dos valores, `true` o `false`, que corresponden a los estados de estar marcado el checkbox o no marcado. Sin embargo, puede resultar útil tener otros valores en caso de estar activos o no, en vez del booleano. Para ello usas estas directivas.

```
<input type="checkbox" ng-model="vm.clarooscuro" ng-true-value="claro" ng-false-value="oscuro" />
```

Directiva `ngChange`

Esta directiva sirve para especificar acciones cuando cambia el estado del checkbox. Pero atención, porque son solo cambios debidos a la interacción con el usuario. Es decir, si mediante Javascript cambiamos el modelo asociado a ese checkbox, cambiándolo de un estado a otro no se activará el evento `ng-change`. La vista seguirá alterando el estado del campo, gracias al mencionado binding, pero la expresión que hayas colocado en `ng-change` no se ejecutará.

```
<input type="checkbox" ng-change="vm.avisar()">
```

Ejemplo de uso de estas directivas en checkbox

Ahora puedes ver un código HTML que trabaja con campos checkbox y que pone en marcha los puntos vistos en este artículo.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Trabajando con checkboxes AngularJS</title>
</head>
<body ng-app="app" ng-controller="appCtrl as vm">

  <p>
```

```
<input type="checkbox" ng-model="vm.activo" ng-change="vm.avisar()">
Este campo es vm.activo y su valor en el modelo es {{ vm.activo }}.
<br />
Tiene además un ng-change asociado con el método vm.avisar().
</p>
<p>
<input type="checkbox" ng-model="vm.clarooscuro" ng-true-value="claro" ng-false-value="oscuro" />
Este campo tiene el value modificado. Ahora vale {{ vm.clarooscuro }}
</p>
<p>
<input type="button" ng-click="vm.activo=true" value="pulsa para cambiar la propiedad del modelo del primer checkbox a true"> Observarás
que aunque el estado pueda cambiar, no se invoca al ng-change de ese primer checkbox.
</p>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular.min.js"></script>
<script>
var app = angular.module("app", [])
app.controller("appCtrl", function(){
    var vm = this;
    //podríamos inicializar valores del modelo
    //vm.activo = false;

    vm.avisar = function(){
        console.log("cambié");
    }
});
</script>
</body>
</html>
```

Observa que al iniciarse por primera vez la página los valores del modelo (definidos en los checkboxes con `ng-model`) no están inicializados. Por ello las expresiones donde se vuelcan estos datos no muestran valor alguno. Luego, cuando cambias el estado de los checkboxes ya se empieza a ver el estado de los elementos. Esta situación podría cambiar solo con inicializar esas propiedades en el controlador. Lo hemos colocado en el código comentado para que se vea bien.

El otro detalle que igual no salta a la vista es la llamada a la función del modelo `vm.avisar()`, colocada en el `ng-change` del primer checkbox, que no se llama cuando se cambia el estado del campo como consecuencia del Javascript. Demostrar eso es el motivo de la existencia del botón.

Hay infinidad de prácticas con checkboxes. Con lo que has visto aquí seguramente sepas desempeñar cualquier situación.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 17/11/2014
Disponible online en <http://desarrolloweb.com/articulos/trabajando-campos-checkbox-angularjs.html>

Filtrar, ordenar y otras variables embutidas en la directiva `ngRepeat`

Usos más detallados de la directiva **ngRepeat** para filtrar, ordenar los elementos y acceder a variables embutidas que nos dan información sobre las repeticiones.

La directiva **ngRepeat** de AngularJS es una de esas cosas que llaman la atención, por la sencillez y rapidez con la que se puede implementar un bucle que nos repite una plantilla HTML, pero sobre todo por su potencia.

La directiva ya la hemos usado en diversos ejemplos anteriormente, incluso tenemos un [artículo anterior donde se introdujeron algunos detalles básicos de ng-repeat](#) para empezar a usarla. Ahora queremos profundizar un poco más en otros usos que resultarán muy útiles para cualquier tipo de aplicación web.

[En la documentación de la directiva ngRepeat](#) encuentras muchos datos útiles, aquí te vamos a resumir algunos y vamos a realizar un ejemplo que lo ponga todo en práctica.



Sintaxis ngRepeat

Nos tenemos que familiarizar primero con la sintaxis usada para especificar el comportamiento del bucle en **ngRepeat**. Aunque ya lo hemos visto en cantidad de ejemplos, comencemos por repasar la sintaxis básica.

```
<p ng-repeat="elem in elementos">
  {{ elem }}
</p>
```

En la directiva indicamos el recorrido a una colección y la variable donde vamos a tener el elemento actual: "elem in elementos". En este caso "elementos" es la colección que vamos a recorrer en el bucle y "elem" es la variable donde vamos a tener el elemento actual en cada repetición.

Pero eso no es lo único que se puede marcar en esta directiva, también podremos especificar otras cosas, colocando el caracter "|" (tubería / pipe) como separador.

```
ng-repeat="elem in elementos | filtros | ordenacion"
```

Técnicamente podemos decir que dentro del HTML de tu repetición tendrás un scope local, donde encontrarás el elemento actual sobre el que estás iterando, así como otras variables útiles que veremos también, como `$index` que nos devuelve el índice actual en el recorrido.

Filtrado en ngRepeat

El filtrado nos sirve para hacer una búsqueda dentro de los elementos de la colección que tenemos en un array o en un objeto en nuestro modelo.

Nota: Por supuesto, la colección está del lado del cliente completa y lo que hace el filter es simplemente definir cuáles elementos de esa colección desean visualizarse en la repetición. Osea, es un filtrado de datos de los que ya disponemos, no un buscador que pueda buscar entre miles de elementos y solo recibir unos pocos en el cliente. Cuando estás filtrando ya tienes esos elementos en la memoria de Javascript en un array u objeto.

Se indica con la palabra filter, seguida por ":" y la cadena o variable donde está la cadena que nos sirve para filtrar.

```
ng-repeat="cerveza in vm.cervezas | filter:'pale' "
```

Esto nos mostrará solo aquellas cervezas que tienen la palabra "pale" escrita por algún lado. Por supuesto, en lugar de un literal de cadena para el filtrado, puedes usar una variable de tu modelo.

```
ng-repeat="cerveza in vm.cervezas | filter:vm.filtroCliente"
```

Ahora utilizarás el contenido de la variable vm.filtroCliente para mostrar aquellos elementos que corresponda.

Orden de los elementos en la repetición

Ahora veamos cómo expresar el orden de visualización de los elementos en la repetición. Para ello usamos la palabra "orderBy", seguido de ":" y el campo sobre el que se debe ordenar. Opcionalmente colocamos después si el orden es ascendente o descendente.

```
ng-repeat="cerveza in vm.cervezas | orderBy:'name':true"
```

En este caso ordenamos los elementos por el campo "name". Lógicamente debe de ser un atributo del objeto que estás recorriendo en la colección. Luego opcionalmente colocamos un booleano para indicar el orden. De manera predeterminada el orden es ascendente. Si el booleano es true, entonces el orden se hace descendente y si es false, entonces es ascendente.

Nuevamente, el orden se indicará habitualmente a partir de variables de Javascript o del modelo de nuestra vista. En un código que podría ser como este:

```
ng-repeat="cerveza in vm.cervezas | orderBy:vm.campo:vm.orden"
```

Variables de la repetición en el scope local

Como hemos dicho existe un scope local que nos expone el elemento actual de nuestra repetición. Pero además tenemos una serie de variables que AngularJS nos ofrece de manera adicional. Son variables que resultarán muy útiles para tus necesidades habituales en recorridos.

Por ejemplo en el scope actual tenemos la variable `$index`, que nos indica el índice actual en la iteración. El primer índice recuerda que es cero, el segundo será uno, etc. Para mostrar ese índice hacemos como con cualquier otro dato que exista en el scope, meterlo dentro de una expresión encerrada con dobles llaves.

```
<p ng-repeat="elem in elementos">
  El elemento actual es {{ $index }} y su valor es {{ elem }}
</p>
```

Junto con el `$index`, este es el conjunto de variables inyectadas automáticamente en el scope local:

- `$index`, numérico. Es el índice de la iteración.
- `$first`, booleano, tendrá el valor `true` para el primer elemento de la iteración.
- `$middle`, booleano, será `true` en los elementos que no sean primero o último.
- `$last`, booleano, solo será `true` en el último elemento.
- `$even`, booleano, será `true` en caso que ese elemento sea impar en la lista de repetición.
- `$odd`, booleano, será `true` en caso que el elemento sea de índice par.

Ejercicio de repetición con `ng-repeat` con filtrado, orden y variables de scope local

Ahora vamos a ver el código de un ejercicio que nos facilite practicar con estos sistemas para personalizar la repetición de un template en AngularJS.

El ejemplo es parecido al ejercicio anterior donde [conocimos JSONP en AngularJS](#), solo que ahora aplicamos las posibilidades de `ngRepeat` vistas en este artículo. Para ello hemos incorporado un par de controles especiales, que nos permiten:

- Escribir un texto para realizar el filtrado
- Seleccionar el campo sobre el que queremos la ordenación y si ésta debe ser ascendente (orden alfabético porque se trata de campos que tienen cadenas) o descendente.

```
<div ng-app="apiApp" ng-controller="apiAppCtrl as vm">
  <h1>Pruebo Ajax con JSONP</h1>
  <p>
    Busca cerveza:
    <input type="text" ng-model="vm.nombre"> <input type="button" value="Buscar" ng-click="vm.buscaCervezas()">
  </p>
  <aside>
    <h2>Filtra:</h2>
```

```

<input type="text" ng-model="vm.filtroCliente">
<h2>Orden</h2>
<p>
  <button ng-click="vm.orden=false">Alfabetico</button>
  <br />
  <button ng-click="vm.orden=true">Contrario</button>
</p>
<p>
  <input type="radio" name="campo" ng-model="vm.campo" value="name"> Nombre
  <br />
  <input type="radio" name="campo" ng-model="vm.campo" value="description"> Descripción
</p>
</aside>
<section>
<ul>
  <li ng-repeat="cerveza in vm.cervezas | filter:vm.filtroCliente | orderBy:vm.campo:vm.orden" ng-class="{even: $even, odd: $odd}">
    <span>{{ $index + 1 }}.- {{cerveza.name}},</span> {{ cerveza.description }}
  </li>
</ul>
</section>
</div>

```

Para el texto del filtrado usamos el INPUT type text que hemos marcado con ng-model="vm.filtroCliente".

Para el orden tenemos dos tipos de controles, diferentes para poder practicar con más cosas. Tenemos un par de botones que nos permiten marcar orden alfabético o contrario, que tienen sus correspondientes ng-click para definir comportamientos. Por otra parte tenemos un par de campos INPUT de radio para definir si queremos que se ordene por el nombre de la cerveza o su campo descripción.

Pero sobre todo te tienes que fijar en el listado de cervezas que realizamos con una lista UL y sus correspondientes LI. Echa un vistazo a la directiva y al template que engloba:

```

<li ng-repeat="cerveza in vm.cervezas | filter:vm.filtroCliente | orderBy:vm.campo:vm.orden" ng-class="{even: $even, odd: $odd}">
  <span>{{ $index + 1 }}.- {{cerveza.name}},</span> {{ cerveza.description }}
</li>

```

Aquí tenemos varios elementos que destacar:

- La repetición se hace con la colección que tenemos en vm.cervezas (que se puebla con una llamada Ajax con JSONP)
- Dentro de nuestro template, el LI y todo su contenido, tendremos disponible la cerveza actual en una variable llamada "cerveza" del scope actual.
- Para el filtrado usamos el contenido del campo de texto con ng-model="vm.filtroCliente".
- Para el orden usamos tanto el valor vm.campo como el vm.orden que se sacan de los botones y los radiobuttons.
- Además puedes ver otra directiva nueva hasta este momento en el [Manual de AngularJS](http://desarrolloweb.com/manuales/manual-angularjs.html) que es ng-

class. Nos sirve para indicarle la class de CSS que queremos aplicar a un elemento HTML. En este caso usamos los valores \$even y \$odd del scope local (datos creados automáticamente por Angular) para colocarle la clase adecuada. De esta manera conseguimos el típico estilo de filas coloreadas como una cebra.

- Por último puedes reparar dentro del contenido de los LI que estamos usando el \$index, que es otra variable del scope local, que nos viene de fábula para numerar las filas de los elementos de la lista.

El Javascript tendrá una pinta como la siguiente:

```
angular
  .module('apiApp', [])
  .controller('apiAppCtrl', controladorPrincipal);

function controladorPrincipal($scope, $http){
  var vm=this;

  vm.orden = false;
  vm.campo = "name";

  var url = "http://api.openbeerdatabase.com/v1/beers.json?callback=JSON_CALLBACK";
  if(vm.nombre){
    url += "&query=" + vm.nombre
  }
  vm.buscaCervezas = function(){
    $http.jsonp(url).success(function(respuesta){
      console.log("res:", respuesta);
      vm.cervezas = respuesta.beers;
    });
  }
}
```

Realmente en este Javascript hay poco que necesites aprender, que no hayas visto ya en artículos anteriores. Te podrá interesar la inicialización de datos en el scope que encuentras en las primeras líneas de la función del controlador.

Aunque para la materia que nos trata no importa demasiado, sí que necesitarás unos estilos CSS para formatear un poco la presentación de ese HTML, de manera que quede bonito y puedas apreciar algunas cosas como los colores de fondo de tipo cebra para los elementos de la lista.

```
body{ font-family: sans-serif;}
li{
  font-size: 0.8em;
  margin-bottom: 10px;
  padding: 10px;
}
li span{
  font-weight: bold;
  display: block;
  font-size: 1.2em;
```

```
}
aside{
  width: 200px;
  float: right;
  padding: 20px;
  display: table-cell;
}
aside h2{
  margin-bottom: 3px;
}
section{
  display: table-cell;
}
li.even{
  background-color: #d5d5d5;
}
li.odd{
  background-color: #d5d5ff;
}
```

Eso es todo de momento, espero que puedas aprovechar ya mismo algunas de estas potentes características de ngRepeat. Como habrás encontrado, esta directiva permite hacer cosas impactantes sin escribir nada de código, algo que apreciarás sobre todo si tienes algo de experiencia en Javascript o librerías como jQuery. Pero OJO, esa misma potencia puede que resulte perjudicial para tu aplicación si se usa sin control, nunca intentes usar ng-repeat con conjuntos masivos de datos, a lo sumo con elementos de 100 en 100, aplicando técnicas como la paginación para reducir los elementos en la vista.

Este artículo es obra de *Alberto Basalo*

Fue publicado por primera vez en 11/11/2014

Disponible online en <http://desarrolloweb.com/articulos/filtrar-ordenar-variables-directiva-ngrepeat.html>

Ajax en Angular

Introducción a los mecanismos para realizar conexiones con el servidor asíncronas por HTTP con AngularJS. Lo que comúnmente conocemos por Ajax, con ejemplos que nos facilitarán entender una de las facetas más potentes de este framework Javascript y el desarrollo frontend basado en las cada vez más populares API REST.

Ajax con AngularJS para acceso a API

Nos introducimos en el uso de Ajax mediante AngularJS. Lo vamos a hacer de una manera muy habitual en el desarrollo con este framework que es accediendo a un servicio web que nos ofrece un API REST.

A la vez que aprendemos técnicas de Ajax vamos a repasar diversas de las prácticas que ya te hemos ido contando a lo largo del [Manual de AngularJS](#).

Ya lo sabes, Ajax es una solicitud HTTP realizada de manera asíncrona con Javascript, para obtener datos de un servidor y mostrarlos en el cliente sin tener que recargar la página entera. Además para los despistados que no lo sepan todavía, tenemos un artículo que te explica muy bien [qué son exactamente las API REST](#).



Service \$http

El servicio \$http (service en inglés, tal como se conoce en AngularJS) es una funcionalidad que forma parte del núcleo de Angular. Sirve para realizar comunicaciones con servidores, por medio de HTTP, a través de Ajax y vía el objeto XMLHttpRequest nativo de Javascript o vía JSONP.

Después de esa denominación formal, que encontramos en la documentación de AngularJS, te debes de quedar por ahora en que nos sirve para realizar solicitudes y para ello el servicio \$http tiene varios tipos de acciones posibles. Todos los puedes invocar a través de los parámetros de la función \$http y además existen varios métodos alternativos (atajos o shortcuts) que sirven para hacer cosas más específicas.

Entre los shortcuts encuentras:

`$http.get()` `$http.post()` `$http.put()` `$http.delete()` `$http.jsonp()` `$http.head()` `$http.patch()`

Tanto el propio `$http()` como las funciones de atajos te devuelven un objeto que con el "patrón promise" te permite definir una serie de funciones a ejecutar cuando ocurran cosas, por ejemplo, que la solicitud HTTP se haya resuelto con éxito o con fracaso.

Nota: Si vienes de jQuery ya habrás podido experimentar las promesas "promise" en las llamadas a la función `$.ajax`, `$.get`, `$.post`, etc. En AngularJS funciona de manera muy similar.

El service `$http` es bastante complejo y tiene muchas cosas para aportar soluciones a las más variadas necesidades de solicitudes HTTP asíncronas. De momento nos vamos a centrar en lo más básico que será suficiente para realizar un ejemplo interesante.

Inyección de dependencias con `$http`

Si vas a querer usar el servicio `$http` lo primero que necesitarás será inyectarlo a tu controlador, o a donde quiera que lo necesites usar. Esto es parecido a lo que mostramos cuando [estábamos practicando con controladores](#) e inyectábamos el `$scope`.

```
angular
  .module('apiApp', [])
  .controller('apiAppCtrl', ['$http', controladorPrincipal] );
```

Como puedes ver, en la función de nuestro controlador, llamada `controladorPrincipal`, le estamos indicando que recibirá un parámetro donde tiene que inyectar el service `$http`.

Al declarar la función recibirás esa dependencia como parámetro.

```
function controladorPrincipal($http){
```

Ahora, dentro del código de ese controlador podrás acceder al servicio `$http` para realizar tus llamadas a Ajax.

Realizar una llamada a `$http.get()`

El método `get()` sirve para hacer una solicitud tipo GET. Recibe diversos parámetros, uno obligatorio, que es la URL y otro opcional, que es la configuración de tu solicitud.

```
$http.get("http://www.example.com")
```

Lo interesante es lo que nos devuelve este método, que es un objeto "HttpPromise", sobre el cual podemos operar para especificar el comportamiento de nuestra aplicación ante diversas situaciones.

Respuesta en caso de éxito

De momento, veamos qué deberíamos hacer para especificarle a Angular lo que debe de hacer cuando se reciba respuesta correcta del servidor.

```
$http.get(url)
  .success(function(respuesta){
    //código en caso de éxito
  });
```

Como puedes ver en este código es que `$http` nos devuelve un objeto. Sobre ese objeto invocamos el método `success()` que sirve para indicarle la función que tenemos que ejecutar en caso de éxito en la solicitud Ajax. Esa función a su vez recibe un parámetro que es la respuesta que nos ha devuelto el servidor.

Nota: En Angular 1.4 cambiaron el estándar de respuesta en caso de éxito o fracaso de la conexión Ajax. El nuevo mecanismo es bastante parecido, con funciones callback que se cargan a partir del método `then()`. Lo tienes descrito con ejemplos en el artículo de [Estándar del patrón promise para los métodos del servicio \\$http en Angular 1.4](#).

Ejemplo completo de solicitud Ajax con \$http

Vistos estos nuevos conocimientos sobre el "service" `$http` estamos en condiciones de hacer un poco de Ajax para conectarnos con un API REST que nos ofrezca unos datos. Esos datos son los que utilizaremos en nuestra pequeña aplicación para mostrar información.

Nota: Vamos a usar un API llamada REST Countries, que puedes encontrar en <http://restcountries.eu/>

Este API nos devuelve JSON con datos sobre los países del mundo. En principio puedes hacer diversas operaciones con los comandos sobre el API, pero vamos a ver solamente unas pocas que nos permiten recibir los datos de los países agrupados por regiones.

Las URL que usaremos para conectarnos al API son como estas:

<http://restcountries.eu/rest/v1/region/africa> <http://restcountries.eu/rest/v1/region/europe>

Puedes abrirlas en tu navegador y observar cómo la respuesta es un conjunto de datos en notación JSON.

Aunque sencilla, esta aplicación ya contiene varias cosillas que para una mejor comprensión conviene ver por separado.

Este es nuestro HTML:

```
<div ng-app="apiApp" ng-controller="apiAppCtrl as vm">
  <h1>Pruebo Ajax</h1>
  <p>
    Selecciona:
    <select ng-model="vm.url" ng-change="vm.buscaEnRegion()">
      <option value="http://restcountries.eu/rest/v1/region/africa">Africa</option>
      <option value="http://restcountries.eu/rest/v1/region/europe">Europa</option>
      <option value="http://restcountries.eu/rest/v1/region/americas">America</option>
    </select>
  </p>
  <ul>
    <li ng-repeat="pais in vm.países">
      País: <span>{{pais.name}}</span>, capital: {{pais.capital}}
    </li>
  </ul>
</div>
```

Puedes fijarte que tenemos un campo SELECT que nos permite seleccionar una región y para cada uno de los OPTION tenemos como value la URL del API REST que usaríamos para obtener los países de esa región.

Aprecia que en el campo SELECT está colocada la directiva ngChange, que se activa cuando cambia el valor seleccionado en el combo. En ese caso se hace una llamada a un método llamado buscaEnRegion() que veremos luego escrito en nuestro controlador.

También encontrarás una lista UL en la que tienes una serie de elementos LI. Esos elementos LI tienen la directiva ngRepeat para iterar sobre un conjunto de países, de modo que tengas un elemento de lista por cada país.

Ahora puedes fijarte en el Javascript:

```
angular
  .module('apiApp', [])
  .controller('apiAppCtrl', ['$http', controladorPrincipal]);

function controladorPrincipal($http){
  var vm=this;

  vm.buscaEnRegion = function(){
    $http.get(vm.url).success(function(respuesta){
      //console.log("res:", respuesta);
      vm.países = respuesta;
    });
  }
}
```

Creamos un module y luego un controlador al que inyectamos el \$http como se explicó al inicio del artículo.

Luego, en la función que construye el controlador, tenemos un método que se llama buscaEnRegion() que es el que se invoca al modificar el valor del SELECT. Ese método es el que contiene la llamada Ajax.

Realmente el Ajax de este ejercicio se limita al siguiente código:

```
$http.get(vm.url).success(function(respuesta){
//console.log("res:", respuesta);
vm.países = respuesta;
});
```

Usamos el shortcut `$http.get()` pasando como parámetro la URL, que sacamos del value que hay marcado en el campo SELECT del HTML. Luego se especifica una función para el caso "success" con el patrón "promise". Esa función devuelve en el parámetro "respuesta" aquello que nos entregó el API REST, que es un JSON con los datos de los países de una región. En nuestro ejemplo, en caso de éxito, simplemente volcamos en un dato del modelo, en el "scope", el contenido de la respuesta.

En concreto ves que la respuesta se vuelca en la variable `vm.países`, que es justamente la colección por la que se itera en el `ng-repeat` de nuestro HTML.

Código completo del ejercicio

Quizás se puede ver y entender mejor el ejercicio en global si vemos el código completo de una sola vez. Pero ya sabes que para aprender bien las cosas debes practicar por tu cuenta, creando ahora tu propio ejemplo Ajax, accediendo a este API REST o a cualquier otra que te apetezca. Verás que es muy sencillo y con poco código puedes hacer cosas bastante más espectaculares de lo que nosotros hemos realizado en este ejemplo.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Probando Ajax en AngularJS</title>
  <style>
    body{ font-family: sans-serif;}
    li{
      font-size: 0.8em;
    }
    li span{
      font-weight: bold;
    }
  </style>
</head>
<body>
<div ng-app="apiApp" ng-controller="apiAppCtrl as vm">
  <h1>Pruebo Ajax</h1>
  <p>
    Selecciona:
    <select ng-model="vm.url" ng-change="vm.buscaEnRegion()">
      <option value="http://restcountries.eu/rest/v1/region/africa">Africa</option>
      <option value="http://restcountries.eu/rest/v1/region/europe">Europa</option>
      <option value="http://restcountries.eu/rest/v1/region/americas">America</option>
    </select>
  </p>
```

```
<ul>
  <li ng-repeat="pais in vm.paises">País: <span>{{pais.name}}</span>, capital: {{pais.capital}}</li>
</ul>

</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular.min.js"></script>
<script>
angular
  .module('apiApp', [])
  .controller('apiAppCtrl', ['$http', controladorPrincipal]);

function controladorPrincipal($http){
  var vm=this;

  vm.buscaEnRegion = function(){
    $http.get(vm.url).success(function(respuesta){
      //console.log("res:", respuesta);
      vm.paises = respuesta;
    });
  }
}

</script>
</body>
</html>
```

En el siguiente artículo explicaremos cómo acceder por Ajax a un JSON pero con JSONP.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 03/11/2014
Disponible online en <http://desarrolloweb.com/articulos/ajax-angularjs-acceso-api-rest.html>

JSONP en AngularJS

Ejercicio Ajax en el que realizamos un acceso a API REST que devuelve información mediante JSONP, realizado con el service \$http de AngularJS.

JSONP es un mecanismo muy útil cuando estamos trabajando con Ajax en Javascript y queremos traernos datos que residen en otro servidor. Existe una problemática por la cual no puedes cargar datos con Javascript que te vengan desde servidores de otros dominios, pues puede darte un error de seguridad si es que el servidor al que te conectas no está configurado para aceptar solicitudes "cross domain".

Ahora no buscamos tanto introducir esta variante de JSON, pues ya la vimos con detalle en el [artículo dedicado a JSONP](#). Simplemente te queremos explicar cómo debes usar JSONP dentro de AngularJS de una manera muy sencilla. Es un excelente ejercicio para practicar con todo lo que llevamos visto en el [Manual de AngularJS](#).



Conexiones HTTP asíncronas en AngularJS

Para realizar solicitudes al servidor asíncronas (lo que se conoce habitualmente por Ajax en el mundo del desarrollo web) con AngularJS necesitamos un servicio llamado `$http`. Este "service" nos lo ofrece Angular de manera completa en su "core" y está preparado para hacer todo tipo de conexiones. Existen varios "shortcuts" que sirven para hacer operaciones típicas, con es el caso de traernos datos ofrecidos por medio de JSONP, así como GET, POST, etc.

En concreto para nuestro ejemplo usaremos el método `jsonp()` de `$http`, que recibe la URL del recurso JSONP al que quieres acceder. Al invocar ese método recibimos un objeto de respuesta sobre el que podemos configurar comportamientos para el caso de que la solicitud tenga éxito, fracaso, etc. Esto se hace con el conocido patrón promise de Javascript.

Nota: En el artículo dedicado a [Ajax en AngularJS](#) explicamos más detalles de este "service" ofrecido por AngularJS.

Función callback del JSONP en la URL de conexión

La verdad es que Angular hace todo el trabajo "sucio" por debajo y tú lo único que necesitas hacer es invocar el método correcto del service `$http`. Así que el método de acceso es prácticamente el mismo que si estuvieras trayendo los datos con un JSON normal.

En este caso la diferencia es que le tienes que indicar un nombre de la función callback de tu JSONP. La restauración del dato se hace de manera automática por AngularJS, lo único que necesitas es que en la URL compuesta de tu JSONP indiques el nombre de la función callback como "JSON_CALLBACK". Eso en JSONP se indica con el parámetro en la URL de conexión llamado "callback", que escribes de la siguiente manera.

Nota: Esa función callback es la que usamos en Javascript del navegador para restaurar los datos del JSONP. Como decimos es algo que realmente no necesitas preocuparte mucho, pues es transparente para ti. Si quieres obtener más información podrías consultar el [artículo sobre JSONP](#).

http://example.com/url.json?callback=JSON_CALLBACK

Ejemplo realizado en AngularJS para traer datos de API REST con JSONP

Ahora vamos a realizar un ejemplo en AngularJS para practicar lo aprendido. De este modo verás que es todo muy sencillo. En este ejemplo traemos datos de cervezas de un API REST pública llamada "Open Beer Database". Los datos los obtenemos por JSONP y puedes ver un ejemplo de conexión y código en esta página de la documentación del API.

<http://openbeerdatabase.com/documentation/examples-jsonp>

En nuestro caso concreto accedemos a cervezas en vez de cervecerías, pero el ejemplo es bien similar. Nuestra URL para obtener datos del API es como esta:

http://api.openbeerdatabase.com/v1/beers.json?callback=JSON_CALLBACK&query=ale

Como puedes ver, en la URL indicamos en el parámetro "callback" el nombre de la función callback que nos pide Angular. Además hay un segundo parámetro llamado "query" con el que podemos expresar unas palabras clave para hacer búsquedas de cervezas que contengan esas palabras.

Sabiendo esto, ahora pasemos a ver nuestro código. Esta es la parte del HTML.

```
<div ng-app="apiApp" ng-controller="apiAppCtrl as vm">
  <h1>Pruebo Ajax con JSONP</h1>
  <p>
    Busca cerveza:
    <input type="text" ng-model="vm.nombre"> <input type="button" value="Buscar" ng-click="vm.buscaCervezas()">
  </p>
  <ul>
    <li ng-repeat="cerveza in vm.cervezas"><span>{{cerveza.name}},</span> {{ cerveza.description }}</li>
  </ul>
</div>
```

Como ves, tenemos un campo de texto donde podemos escribir un dato y un botón de buscar. Al darle a buscar llamamos a un método de nuestro "scope" para traernos las cervezas que tengan el texto escrito en el campo de texto, ya sea en su nombre o descripción.

Luego tenemos un bucle definido con ng-repeat en el que recorremos una colección de cervezas.

Ahora puedes ver la parte del Javascript:

```
angular
  .module('apiApp', [])
  .controller('apiAppCtrl', controladorPrincipal);

function controladorPrincipal($scope, $http){
  var vm=this;

  vm.buscaCervezas = function(){
    var url = "http://api.openbeerdatabase.com/v1/beers.json?callback=JSON_CALLBACK";
    if(vm.nombre){
      url += "&query=" + vm.nombre;
    }
  }
}
```

```
}  
$http.jsonp(url).success(function(respuesta){  
    console.log("res:", respuesta);  
    vm.cervezas = respuesta.beers;  
});  
}  
}
```

Bien, supongo que estos códigos ya te irán sonando y en concreto este ejercicio es muy parecido al anterior en el que conocimos Ajax. En la parte importante, nuestro controlador, apreciarás que tenemos un método llamado `buscaCervezas()` que es el que se encarga de hacer todo el trabajo.

En ese método primero construimos la URL para acceder al API, agregándole el nombre de la cerveza que quieres buscar.

Luego accedemos por medio de `$http.jsonp()` a la URL construida y definimos una función que se ejecutará en caso de éxito de la conexión Ajax (`success`). en esa función simplemente volcamos las cervezas encontradas en el scope, con lo que se actualiza automáticamente la vista en el bloque donde teníamos la directiva `ng-repeat` de nuestro HTML.

Con esto es todo. Solo te queda practicar lo aprendido por tu cuenta para no dejarte un detalle sobre las conexiones JSONP en AngularJS.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 06/11/2014
Disponible online en <http://desarrolloweb.com/articulos/jsonp-angularjs-ejercicio-ajax-api-rest.html>

Operación post HTTP con \$http en AngularJS

Cómo realizar una operación POST con Ajax, para enviar datos al servidor por medio del HTTP con la librería AngularJS y el service \$http.

Hemos visto cómo realizar operaciones básicas de Ajax con AngularJS, en las que [recibimos información del servidor, por medio de get](#). Ahora veamos también cómo se enviarían datos al servidor por medio de la operación post del HTTP. Como siempre las cosas con AngularJS se hacen muy sencillas.

El servicio ("service" en la terminología de Angular) `$http` nos ofrece una serie de métodos "shortcut" enfocados en realizar las operaciones típicas implementadas dentro del protocolo HTTP. Para enviar datos post disponemos de `$http.post()`. En ese método podemos enviar como parámetro, aparte de la URL del servidor donde haremos el post, un objeto con los datos que se desean enviar.



```
$http.post("recibe.php", {uno: 1, fruta: "manzana"});
```

Como ves, la primera dirección es la URL relativa donde enviar los datos (también podría ser absoluta) y el segundo parámetro son los datos que queremos enviar al destino.

Este método devuelve como resultado un objeto sobre el que podemos implementar con el patrón promesa algunas operaciones en diversas situaciones con respecto a esa conexión con el servidor, algo que ya viste en lo ejemplos de Ajax anteriores. Lo que vamos a aprender de momento es realizar acciones en caso de éxito y para ello tenemos que indicarlo con una función en "success".

```
var conAjax = $http.post("recibe.php", {uno: 1, fruta: "manzana"});  
conAjax.success(function(respuesta){  
    console.log(respuesta);  
});
```

En la función que asociamos al caso success, como ya habrás visto muchas veces, recibimos un parámetro con la respuesta que nos devuelve el servidor. En este caso simplemente la volcamos a la consola de Javascript.

Nota: Este código lo encontrarás habitualmente encadenando llamadas, sin necesidad de declarar la variable "conAjax". Luego haremos ejemplos que usen esa común forma de codificar.

Recibir los datos en el servidor

Los datos que estás enviando por post Angular los empaqueta como JSON y te llegarán al servidor, aunque no por el método común POST, en pares clave/valor como quizás estás acostumbrado. En realidad nos lo envía como content-type, "application/json" en un único objeto, lo que es útil porque nos acepta datos más complejos, con anidación de objetos JSON.

Nota: Para que nos entendamos, en un lenguaje como PHP, cuando enviamos datos por post los recoges con el array \$_POST. Ese array no aceptaría diversos niveles de anidación para recibir datos complejos, osea, podemos tener claves con valores simples. Pero se nos queda un poco corto para aplicaciones modernas. En PHP si accedes a \$_POST encontrarás que el array está vacío.

Este asunto de los niveles de anidación en los datos que se envían con JSON se ve por ejemplo en este código, perfectamente válido y habitual.

```
$http.post("recibe2.php", {
  nombre: "Miguel",
  fechaNacimiento: "21/02/1975",
  sitiosPreferidos: [
    "DesarrolloWeb.com",
    "Guiarte.com"
  ],
  direccion: {
    calle: "De la alegría",
    numero: 18,
    ciudad: "Villadigital"
  }
})
.success(function(respuesta){
  console.log(respuesta);
});
```

Ese objeto complejo JSON lo recibirás en el servidor tal cual. El código para recogerlo dependerá de tu lenguaje de servidor. Por ejemplo en PHP lo haría de esta manera:

```
file_get_contents("php://input")
```

Eso nos devolvería una cadena de texto, que si quieres volcar a un objeto nativo de PHP usarás la función `json_decode()`. Tu código podría quedarte parecido a este:

```
$objDatos = json_decode(file_get_contents("php://input"));
```

A partir de ese momento encontrarás que puedes acceder a los datos del JSON recibido como estás acostumbrado en el trabajo con objetos PHP.

```
echo $objDatos->nombre;
echo $objDatos->sitiosPreferidos[0];
echo $objDatos->direccion->calle;
```

Enviar un formulario por POST

Si lo que quieres es enviar los datos que se encuentran en un formulario por POST al servidor, para recibirlos en un JSON, la verdad es que el procedimiento es bien parecido a lo que has visto. Simplemente tendremos que crear nuestro modelo con los datos del formulario, algo que hace Angular por ti agregando la directiva `ngModel`, y enviarlo por `$http.post()`.

Echa un vistazo a este formulario:

```
<div ng-app="app" ng-controller="appCtrl as vm">
  <h1>Pruebo Ajax</h1>
  <section>
    <form ng-submit="vm.enviar()">
      Nombre: <input type="text" ng-model="vm.fdatos.nombre">
      <br>
      Edad: <input type="text" ng-model="vm.fdatos.edad">
      <br>
      <input type="submit" value="Enviar">
    </form>
  </section>
</div>
```

Observa que en ng-model hemos volcado los campos del formulario dentro de un objeto llamado "fdatos". Osea, en el modelo de la vista "vm" tenemos un objeto "datosf" y dentro ya encontraremos los datos de nuestro formulario.

También repara en la directiva ngSubmit que hemos colocado en la etiqueta FORM. Verás que la hemos asociado con una función de nuestro modelo: vm.enviar(). En esa función que veremos a continuación es donde debes escribir el código para poder enviar el formulario.

Nota: Esto no estás obligado a hacerlo así necesariamente, porque puedes crear la estructura que desees en el modelo. Simplemente nos resultará más cómodo colocar directamente los datos vinculados a un objeto de datos del formulario. Al servidor por post no vamos a enviar todo el modelo de la vista, sino únicamente los datos del formulario. Como esta estructura ya nos permite tener los datos del formulario en un objeto independiente, nos ahorrará el tener que hacerlo "a mano". Enseguida lo verás mejor.

Para que esto funcione en tu Javascript debes inicializar "fdatos" como un objeto, aunque sea vacío. Por ello en tu controller deberías incluir este código.

```
vm.fdatos = {};
```

Es un código parcial, luego lo verás en el contexto del controlador completo. Observa que en tu "vm" has inicializado el objeto "fdatos" con un literal de objeto vacío, expresado con las dos llaves.

Esto ya nos deja en muy buena situación para enviar ese formulario de una manera muy limpia. Ahora te mostramos el código completo para crear nuestro controlador.

```
angular
  .module('app', [])
  .controller('appCtrl', ['$http', controladorPrincipal]);

function controladorPrincipal($http){
  var vm=this;
```

```
//inicializo un objeto en los datos de formulario
vm.fdatos = {};

// declaro la función enviar
vm.enviar = function(){
    $http.post("recibe-formulario.php", vm.fdatos)
        .success(function(res){
            console.log(res);
            //por supuesto podrás volcar la respuesta al modelo con algo como vm.res = res;
        });
}
```

Con esto creo que lo tendrás todo claro para poder enviar datos a un servidor por medio de post, datos que podrán ser todo lo complejos que necesites en tu aplicación. Como puedes comprobar la llamada a `$http.post()` te hace todo el trabajo de la solicitud HTTP por medio de Ajax, por lo que solo tendrás que programar el comportamiento específico para tus necesidades, tanto en el cliente como en el servidor.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 24/11/2014
Disponible online en <http://desarrolloweb.com/articulos/operacion-post-http-angularjs.html>

Promise estándar `then()` de `$http` en AngularJS 1.4

Explicaciones sobre el nuevo estándar promise para llamadas Ajax en Angular 1.4, usando el método `then()`.

En el [Manual de AngularJS](#) hemos explicado Ajax en diversos artículos, sin embargo unos cambios en las últimas versiones de este framework Javascript nos obligan a hacer algunas actualizaciones relacionadas con el estándar promise.

Cuando realizamos una conexión por Ajax siempre tardará un poco en recibirse la respuesta. Gracias al comportamiento asíncrono de Javascript, no bloqueante ante situaciones de espera, se tiene que usar el patrón promise para indicar las acciones a realizar cuando se produce esa respuesta del servidor.

En los [artículos anteriores dedicados a Ajax](#) se mostró el funcionamiento del sistema de promesas a través de los métodos `success()` y `error()`. Éstos ahora están obsoletos, por lo que debemos empezar a acostumbrarnos a usar el nuevo estándar con el método `then()`.

Nota: a pesar que los métodos `success()` y `error()` están obsoletos (deprecated) siguen funcionando, para asegurar que no da errores el código creado para versiones anteriores de AngularJS. Son considerados como "legacy methods" y no conviene usarlos porque en un futuro se dejará de dar soporte.



Cómo obtener el objeto promise

Por refrescar la memoria, vamos a ver el código de una llamada Ajax, usando el service `$http` con el método shortcut `$http.get()`. Éste método nos devolverá el mencionado objeto "promesa".

```
var promesa = $http.get("https://restcountries.eu/rest/v1/all");
```

Aunque en seguida veremos que generalmente el objeto no es necesario guardarlo en una variable, sino encadenar directamente la configuración del "promise".

Método then()

Sobre el objeto promesa podemos invocar el método `then()`. Éste nos permite definir dos funciones con las cuales podremos indicar las acciones a realizar, tanto para el caso de suceso en la solicitud como para el caso de error.

Lo común sería utilizar un sistema de encadenamiento de métodos, que nos permita recibir el objeto promesa y a continuación invocar su método `then()` para realizar las acciones oportunas cuando se reciba la respuesta del servidor.

```
$http.get("http://example.com/url/ajax")
  .then(function(res){
    // acciones a realizar cuando se recibe respuesta con éxito
  }, function(res){
    // acciones a realizar cuando se recibe una respuesta de error
  });
```

Cómo has podido apreciar, el método `then()` recibe dos parámetros que configuramos con dos funciones anónimas. La primera de las funciones sirve para indicar las acciones que quieres realizar cuando recibes del servidor una respuesta con éxito, es decir, una respuesta con un estatus del protocolo HTTP que considere como "todo correcto", status 200 por ejemplo. La segunda de las funciones sirve para indicar las acciones que se quiere realizar al recibir una respuesta de error, con un estatus tal qué 404, de página no encontrada, o 500, de error de ejecución del servidor, etc.

Parámetro respuesta en las funciones anónimas

Como novedad, las funciones anónimas que asignamos al método `then()`, reciben un objeto respuesta.

```
.then(function(res){
```

Este objeto tiene una serie de atributos con los datos de la respuesta que hemos recibido del servidor. Por ejemplo podremos encontrar el atributo "data", que nos devolvería el objeto nativo de Javascript creado a partir del JSON que se recibe en el cuerpo del response. También podremos encontrar por ejemplo un atributo "status" con el valor exacto del status de respuesta, 200, 404 o el que sea.

Para observar ese nuevo objeto respuesta te recomiendo hacer un `console.log()` y así podrás apreciar todos los atributos que nos entrega Angular para indicarnos exactamente cómo ha ido la solicitud Ajax. No obstante ahora vamos a poner un pequeño ejemplo donde podrás ver cómo podremos utilizar este objeto de respuesta para producir una salida adecuada es una solicitud Ajax con angularjs.

Ejemplo completo Ajax en AngularJS 1.4

Comencemos primero mostrando el código HTML de nuestro ejemplo. Nos estamos conectado con un API pública llamada REST Countries, que nos facilita información diversa sobre países del mundo.

Esta página mostrará en una lista todos los países que nos devuelva este API. Además se mostrará al lado la capital del país. El propio código HTML tienes enlaces a REST Countries, por si quieres echarle un vistazo a esta API muy interesante para realizar ejemplos Ajax con angular sin tener que preocuparte por la autenticación.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Ajax con un API REST</title>
  <link rel="stylesheet" href="../estilo-basico.css">
</head>
<body ng-app="paísesApp" ng-controller="PaísesAppController">
  <h1>Países del mundo</h1>
  <p>
    Vamos a usar el servicio web llamado "Rest Countries".
    <a href="https://restcountries.eu/">https://restcountries.eu/</a>
  </p>
  <p>
    La URL <a href="https://restcountries.eu/rest/v1/all">https://restcountries.eu/rest/v1/all</a> devuelve todos los países del mundo.
  </p>
  <div ng-show="países.length == 0">Cargando países...</div>
  <h2 ng-show="países.length">Se encontraron {{países.length}} en el mundo</h2>
  <ol>
    <li ng-repeat="pais in países">{{pais.name}} - {{pais.capital}}</li>
  </ol>

  <script src="../angular.min.js"></script>
  <script src="paísesApp.js"></script>
</body>
</html>
```

Ahora vamos a mostrar el código JavaScript de nuestro "Module". Apremiarás que tienes un controlador que es donde realizamos la solicitud Ajax para poblar array de países con el que realizar el listado en la página.

```
angular
  .module("paísesApp", [])
  .controller("PaísesAppController", ["$scope", "$http", function($scope, $http){
    $scope.países = [];
    $http.get("https://restcountries.eu/rest/v1/all")
      .then(function(res){
        $scope.países = res.data;
      }, function(res){
        $scope.países = [{name: "Error!! " + res.status}];
      });
  });
```

Fíjate en la primera función anónima que enviamos al método then(), cómo hacemos uso del atributo data de la respuesta para asignarlo al array de países del scope.

La segunda función anónima indicada en caso de error utiliza la respuesta para extraer el estatus recibido (atributo status). No sería la mejor manera de mostrar un error pero nos sirve como alternativa para este pequeño ejemplo.

Conclusión

Con esto terminamos este ejercicio en el cual hemos visto el nuevo estándar de promesas para solicitudes Ajax que utilizamos en las últimas versiones de angularjs. La verdad es que no tiene ninguna dificultad por lo que entendemos que no hace falta mucha más explicación para poder empezar a usarlo y adaptarnos API de las nuevas versiones de este framework Javascript.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 02/10/2015
Disponible online en <http://desarrolloweb.com/articulos/promise-estandar-ajax-angularjs.html>

Crear un API REST, 5 minutos con json-server

Crear en 5 minutos un API REST, con fines didácticos, ideal para aprender desarrollo frontend con un framework Javascript del lado del cliente, usando json-server.

En este artículo vamos a ver cómo podemos crear un API REST en menos de 5 minutos, que podremos usar con fines didácticos o para prototipado, ideal cuando estamos aprendiendo a desarrollar con un framework Javascript del lado del cliente.

La idea es conseguir un API perfectamente funcional, que admita las operaciones típicas de cualquier API REST estándar, ofreciendo las típicas operaciones través de los distintos verbos del HTTP, pero sin tener que invertir horas de nuestro tiempo para aprovisionarlo.

Para poder ser tan rápidos usaremos un sencillo archivo JSON como fuente de datos, donde podremos generar nuestro modelo de datos, con distintas entidades, y datos de prueba. Como servidor REST usaremos un programa llamado "json-server", desarrollado con NodeJS que nos ofrece la posibilidad de tener un servidor web que responde a las operaciones típicas de las API.

El API REST que vamos a crear será perfectamente funcional, es decir, realizará todas las operaciones posibles sobre nuestros datos (lecturas, modificaciones, inserciones y borrados), almacenando los datos en el archivo JSON. La persistencia de los datos se realiza en el propio archivo de texto JSON, de este modo, en futuros accesos a la aplicación web, el API recordará su estado manteniendo toda la actividad que se haya realizado a lo largo del tiempo.



Al final del artículo verás además un vídeo donde se explica todo el proceso y se muestra cómo realizarlo en pocos minutos.

Nota: Si no tienes claro lo que es REST te recomiendo que te leas el artículo [Qué es REST](#) de Eduard Tomàs que lo explica muy bien y muy rápido. Incluso si tienes una idea de lo que es un API REST y quieres completar la información y saber más detalles de este tipo de recursos, la lectura te vendrá muy bien.

Descargar json-server

El primer paso para tener nuestra API es descargar e instalar json-server. Este programa es un paquete de NodeJS que instalaremos vía npm.

Nota: Si no tienes NodeJS deberás instalarlo en tu ordenador. Es muy sencillo entrando en <https://nodejs.org> y siguiendo las instrucciones para instalación. También te recomendamos el [Manual de NodeJS de DesarrolloWeb.com](#), donde encontrarás información útil sobre "Node". El gestor de paquetes "npm" se instala automáticamente durante la instalación de NodeJS en tu máquina.

Realizaremos el siguiente comando en nuestro terminal:

```
npm install -g json-server
```


El gestor de paquetes npm nos instalará json-server con todas sus dependencias. Mientras tanto, podemos ir realizando el siguiente paso.

Crear un archivo JSON con los datos de nuestra API

En cualquier carpeta de nuestro ordenador debemos crear el archivo JSON que va a servir de origen de datos para nuestra API. Los datos serán aquellos que necesitemos en nuestra aplicación y en el propio JSON se puede introducir de inicio un conjunto de datos de prueba.

Nota: Puedes [saber algo más de JSON en este artículo](#). En realidad no es más un archivo de texto plano, que guardarás con codificación UTF-8. La notación para definir los datos la realizamos como si fuera un objeto Javascript. Al trabajar con JSON no se realiza una definición del modelo de datos como se hace en una base de datos relacional, ya que este modelo de datos se define a través de los propios datos.

Podemos ver un juego de datos de muestra, en el siguiente código JSON, que almacena películas y un conjunto de clasificaciones de películas.

```
{
  "peliculas": [
    {
      "id": 1,
      "nombre": "El sexto sentido",
      "director": "M. Night Shyamalan",
      "clasificacion": "Drama"
    },
    {
      "id": 2,
      "nombre": "Pulp Fiction",
      "director": "Tarantino",
      "clasificacion": "Acción"
    },
    {
      "id": 3,
      "nombre": "Todo Sobre Mi Madre",
      "director": "Almodobar",
      "clasificacion": "Drama"
    },
    {
      "id": 4,
      "nombre": "300",
      "director": "Zack Snyder",
      "clasificacion": "Acción"
    },
    {
      "id": 5,
      "nombre": "El silencio de los corderos",
      "director": "Jonathan Demme",
      "clasificacion": "Drama"
    }
  ]
}
```

```
{,
{
  "id": 6,
  "nombre": "Forrest Gump",
  "director": "Robert Zemeckis",
  "clasificacion": "Comedia"
},
{
  "id": 7,
  "nombre": "Las Hurdes",
  "director": "Luis Buñuel",
  "clasificacion": "Documental"
}
],
"clasificaciones": [
{
  "nombre": "Drama",
  "id": 1
},
{
  "nombre": "Comedia",
  "id": 2
},
{
  "nombre": "Documental",
  "id": 3
},
{
  "nombre": "Acción",
  "id": 4
}
]
}
```

Este archivo JSON tendrá típicamente una extensión .json. En nuestro caso lo podríamos guardar como "películas.json".

Arrancar el servidor del API REST

Como tercer y último paso para tener nuestro API funcionando debemos arrancar el servidor del API, es decir, nuestro recién instalado json-server. Esto también lo hacemos desde el terminal, con el siguiente comando.

```
json-server --watch peliculas.json
```

Enseguida veremos que nos aparece una serie de mensajes indicando cómo podríamos acceder a nuestro origen de datos, a través de las URL del servidor, o las URL de los recursos o entidades generadas en el JSON.

```
\{^_^}/ hi!
```

```
Loading peliculas.json  
Done
```

Resources

```
http://localhost:3000/peliculas  
http://localhost:3000/clasificaciones
```

Home

```
http://localhost:3000
```

```
Type s + enter at any time to create a snapshot of the database  
Watching...
```

Acceder al API

Ahora ya solo nos queda acceder al API. Realmente podrás entrar en la "home" del servidor json-server para tu API a través de una URL como esta:

```
http://localhost:3000
```

En esa página encontrarás enlaces a los distintos recursos de tu API. Si los abres verás que te devuelve datos JSON como la mayoría de las API REST. Las operaciones que podrás realizar sobre el API son las típicas y se opera básicamente mediante los verbos del HTTP. Realmente aquí no difiere en nada de cualquier otro API que puedas usar.

Además, el API está perfectamente habilitada para invocarla desde otros dominios gracias a tener habilitado CORS, o si se desea usando [JSONP](#).

En resumen, en menos de 5 minutos tenemos un API funcionando que nos vendrá perfecto para realizar nuestra programación Javascript del lado del cliente con acceso a datos de un API. Como hemos dicho, este servidor de JSON está más enfocado a fines didácticos o también al prototipado de un proyecto web o App para móviles.

Objeto de estudio aparte sería el acceso a esos datos, que generalmente harás con Ajax, usando un framework como [AngularJS](#) o una librería como [BackboneJS](#), o al menos con [jQuery](#).

Vídeo de la creación del API

A continuación puedes ver todo el proceso de creación del API REST en vídeo.

Para ver este vídeo es necesario visitar el artículo original en: <http://desarrolloweb.com/articulos/crear-api-rest-json-server.html>

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado por primera vez en 27/11/2015

Disponible online en <http://desarrolloweb.com/articulos/crear-api-rest-json-server.html>

Vistas y sistema de Routing en Angular

En resumen, las vistas nos permiten separar partes de nuestra interfaz en partes distintas y con el sistema de routing AngularJS es capaz de saber qué vistas debe mostrarte atendiendo a la URL que la aplicación esté mostrando. Veremos en detalle cómo trabajar con múltiples vistas y rutas profundas en una aplicación desarrollada con este framework Javascript.

Módulo ngRoute para crear rutas e intercambiar vistas en AngularJS

Explicaciones sobre ngRoute, el módulo de AngularJS que nos permite crear rutas profundas en nuestra aplicación e intercambiar vistas dependiendo de la ruta.

Hasta ahora en el [Manual de AngularJS](#) hemos visto muchas cosas interesantes, sin embargo nos hemos limitado a hacer aplicaciones en las que solo teníamos una ruta y una vista. Sin embargo, cuando se vayan complicando los requisitos de nuestras aplicaciones podremos necesitar ampliar estas posibilidades.

Lo que vamos a aprender ahora es a crear rutas distintas dentro de nuestra aplicación, de modo que tengamos varias URL que muestran vistas distintas. Quizás te preguntes ¿Si AngularJS está especialmente indicado para hacer aplicaciones de una sola página (concepto que se conoce con el nombre de "Single Page Application", SPA), por qué ahora nos muestras la posibilidad de crear varias URL? ¿No era una única página lo que queríamos hacer?



Pues bien, podemos tener ambas cosas. Una aplicación de una sola página, pero que es capaz de representar URL distintas, simulando lo que sería una navegación a través de la aplicación, pero sin salirnos nunca de la página inicial. Esto nos sirve para varias cosas, entre otras:

- **Memorizar rutas profundas** dentro de nuestra aplicación. Podemos contar con enlaces que nos lleven a partes internas (deeplinks), de modo que no estemos obligados a entrar en la aplicación a través de la pantalla inicial.
- **Eso facilita también el uso natural del sistema de favoritos (o marcadores) del navegador, así como el historial.** Es decir, gracias a las rutas internas, seremos capaces de guardar en favoritos

un estado determinado de la aplicación. A través del uso del historial del navegador, para ir hacia delante y atrás en las páginas, podremos navegar entre pantallas de la aplicación con los botones del navegador.

- **Mantener vistas en archivos independientes**, lo que reduce su complejidad y administrar los controladores que van a facilitar el procesamiento dentro de ellas.

Cómo son las rutas internas de la aplicación

Para que nos entendamos, en nuestra aplicación vamos a poder tener varias URL. Podrán tener una forma como esta:

<http://example.com/index.php> <http://example.com/index.php#/seccion>
http://example.com/index.php#/pagina_interna

Son simples ejemplos, lo importante es fijarse en el patrón "#/". Podrás darte cuenta que estas URL corresponden con la misma página, todas, pues usan el carácter "#" que nos sirve para hacer lo que se llaman "enlaces internos" dentro del mismo documento HTML. Como sabemos, la "almohadilla" ("gato", "numeral", "sostenido" o como la llares en tu país), sirve para hacer rutas a anclas internas, zonas de una página.

Cuando le pidamos al navegador que acceda a una ruta creada con "#" éste no va a recargar la página, yéndose a otro documento. Lo que haría es buscar el ancla que corresponda y mover el scroll de la página a ese lugar.

Nota: Éste es el comportamiento de cualquier navegador, no estamos apuntando nada que tenga que ver con Javascript o con AngularJS. Angular, así como otros frameworks Javascript MVC se aprovechan de esta característica para implementar el sistema de enrutado.

En el caso de AngularJS no habrá un movimiento de scroll, pues con Javascript se detectará el cambio de ruta en la barra de direcciones para intercambiar la vista que se está mostrando.

Por ello, volviendo a la pregunta de antes: "si la posibilidad de crear varias rutas dentro de una aplicación contradice el sistema de Single Page Application", observamos que realmente no son páginas distintas, sino que es la misma página. Lo que estamos haciendo es "simular" la navegación por varias URL cuando realmente es la misma, con enlaces internos.

Instalar ngRoute

El módulo ("module" en la terminología anglosajona de Angular) ngRoute es un potente paquete de utilidades para configurar el enrutado y asociar cada ruta a una vista y un controlador, tal como hemos dicho. Sin embargo este módulo no está incluido en la distribución de base de Angular, sino que en caso de pretender usarlo tenemos que instalarlo y luego inyectarlo como dependencia en el módulo principal de nuestra aplicación.

Como no lo tenemos en el script básico de Angular, debemos instalarlo "a mano". lo conseguimos incluyendo el script del código Javascript del módulo ngRoute.

```
<script src="angular-route.js"></script>
```

Nota: Ojo que esta ruta no debes copiar y pegarla tal cual. Muchas veces usarás el CDN correspondiente y deberás asegurarte de estar usando en el módulo ngRoute de la misma versión de AngularJS que cargaste inicialmente. Otro detalle es que este script lo tienes que incluir después de haber incluido el script del core de AngularJS.

Inyección de dependencias

El segundo paso sería inyectar la dependencia con ngRoute en el módulo general de nuestra aplicación. Esto lo hacemos en la llamada al método `module()` con el que iniciamos cualquier programa AngularJS, indicando el nombre de las dependencias a inyectar en un array.

```
angular.module("app", ["ngRoute"])
```

Hasta ahora este array de dependencias, cuando llamábamos a `angular.module()`, estaba siempre vacío. A medida que se compliquen nuestras aplicaciones podremos necesitar inyectar más cosas. Por ejemplo, las directivas creadas por terceros desarrolladores también las inyectarás de esta manera en tu módulo principal de la aplicación. Eso lo veremos más adelante.

Configurar el sistema de enrutado con \$routeProvider

El sistema de enrutado de AngularJS nos permite configurar las rutas que queremos crear en nuestra aplicación de una manera declarativa. Aunque sea un componente bastante complejo internamente, podemos configurarlo de una manera ciertamente sencilla.

`$routeProvider` tiene un par de métodos. El primero es `when()`, que nos sirve para indicar qué se debe hacer en cada ruta que se desee configurar, y el método `otherwise()` que nos sirve para marcar un comportamiento cuando se intente acceder a cualquier otra ruta no declarada.

Esta configuración se debe realizar dentro del método `config()`, que pertenece al modelo. De hecho, solo podemos inyectar `$routeProvider` en el método `config()` de configuración.

```
angular.module("app", ["ngRoute"])
  .config(function($routeProvider){
    //configuración y definición de las rutas
  });
```

Las rutas las configuras por medio del método `when()` que recibe dos parámetros. Por un lado la ruta que se está configurando y por otro lado un objeto que tendrá los valores asociados a esa ruta. De momento vamos a conocer solo éstos que serían los fundamentales para empezar.

- "controller", para indicar el controlador

- "controllerAs", el nombre con el que se conocerá el scope dentro de esa plantilla
- "templateUrl", el nombre del archivo, o ruta, donde se encuentra el HTML de la vista que se debe cargar cuando se acceda a la ruta.

Podemos ver un ejemplo completo de configuración de rutas.

```
angular.module("app", ["ngRoute"])
.config(function($routeProvider){
  $routeProvider
    .when("/", {
      controller: "appCtrl",
      controllerAs: "vm",
      templateUrl: "home.html"
    })
    .when("/descargas", {
      controller: "appCtrl",
      controllerAs: "vm",
      templateUrl: "descargas.html"
    })
    .when("/opciones", {
      controller: "appCtrl",
      controllerAs: "vm",
      templateUrl: "opciones.html"
    });
});

.controller("appCtrl", function(){
  //código del controlador (lo estoy usando en todas las rutas, en este sencillo ejemplo)
});
```

Hemos encadenado varias llamadas a métodos `when()` sobre el `$routeProvider`, para definir cada una de las rutas que tendremos en nuestra pequeña aplicación.

Nota: Observa que en cada una de las rutas tenemos definido el mismo controlador. Esto no tiene por qué ser así, podrías perfectamente tener un controlador distinto para cada ruta del sistema.

Para finalizar, podemos ver el código HTML de una página donde usaríamos estas rutas.

```
<body ng-app="app">
  <nav>
    <ul>
      <li><a href="#">Datos personales</a></li>
      <li><a href="#/descargas">Descargas</a></li>
      <li><a href="#/opciones">Opciones de cuenta</a></li>
    </ul>
  </nav>
  <hr />
```

```
<div ng-view></div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular-route.js"></script>
<script src="app.js"></script>

</body>
```

Echa un vistazo a los enlaces que hay dentro de la etiqueta NAV. Esos son los enlaces que provocarían la navegación en el enrutado definido en la anterior configuración. Observarás que los href de los enlaces están asociados a las mismas rutas definidas en el `$routeProvider`. Por tanto, pulsando cada enlace Angular nos mostrará la vista indicada en la declaración que hemos mostrando antes en el `config()` del `$routeProvider`.

Tienes también que fijarte en la división DIV que hay más abajo que tiene la directiva `ngView`. En principio no necesitas indicar más cosas como valor de esa directiva. AngularJS ya sabe que las vistas las debe desplegar en ese contenedor.

Solo a modo de curiosidad, fíjate que en el HTML no hemos definido ningún controlador, osea, no está la directiva `ng-controller` por ninguna parte. Sin embargo, en las vistas internas sí que se está asociando un controlador, porque se ha declarado en el `$routeProvider`.

Nota: No queremos adelantarnos, pero si se te ocurre jugar con este prototipo y creas un poco de funcionalidad en el controlador que estamos usando en las vistas comprobarás que los datos se inicializan cada vez que accedemos a las vistas de nuevo. Esto lo explicaremos más adelante, así que no te preocupes si no has entendido todavía a qué nos estamos refiriendo.

Vistas independientes

Para acabar solo queda comentar el código de las vistas independientes. Tal como se definió en el `$routeProvider`, existe un código para cada una de las vistas, alojado en archivos HTML independientes. La vista alojada en `"/` se llama `"home.html"`, la vista de la ruta `"/descargas"` está en el archivo `"descargas.html"`, etc. Esos archivos están en el mismo directorio que la vista principal, pero realmente podrían estar en cualquier otra ruta que especifiquemos.

El código que coloques en las vistas es indiferente. Simplemente pondrás HTML, directivas y todo lo que venimos usando hasta el momento en las vistas. Escribe el HTML que gustes, de momento no tiene más importancia.

Este artículo es obra de *Alberto Basalo*
Fue publicado por primera vez en 03/12/2014
Disponible online en <http://desarrolloweb.com/articulos/ngroute-crear-rutas-intercambiar-vistas-angularjs.html>

Introducción a \$location y primeros controladores en paralelo

Ejercicio donde tenemos en marcha dos controladores ejecutándose en paralelo en dos partes de la página, así como varias vistas. Conocemos también \$location de AngularJS.

Es común usar más de un controlador en una aplicación AngularJS. Hasta ahora no lo habíamos necesitado pero para separar nuestro código en controladores puede aportar diversos beneficios, en el caso que tenga algún sentido esa separación.

En el ejemplo que os traemos queda bien claro que tiene sentido separar el código en dos controladores distintos, porque están pensados para hacer cosas muy específicas y distintas entre sí. Observarás que separar la complejidad del programa en diversas partes, provoca que cada una sea más simple y fácil de entender. Todo ello redundará en un mantenimiento más sencillo, lo que al final se traduce en menos trabajo y mayor rentabilidad.



Para hacer este ejercicio hemos seguido el prototipo realizado en [el artículo anterior dedicado al ngRoute](#), por lo que si no lo has leído ya, es importante que lo hagas. En aquel ejemplo vimos cómo implementar enlaces "profundos", en una barra de navegación, que nos llevan a distintas vistas dentro de la misma página.

Ahora vamos a mejorar el comportamiento de nuestra barra de navegación. Aislaremos ese comportamiento en un nuevo controlador independiente. Con ello tendremos dos controladores bien separados:

- Un controlador para la barra de navegación de secciones
- Un controlador para las vistas de mi aplicación

Como puedes ver, son dos áreas distintas de la aplicación y por lo tanto tiene sentido usar controladores distintos. El controlador de la navegación sirve simplemente para implementar un comportamiento en mi barra de navegación, mientras que el controlador de las vistas independientes serviría para mantener la funcionalidad de esas vistas.

Echemos un vistazo en el código HTML de este ejercicio

```
<body ng-app="app">
  <nav ng-controller="navCtrl as nav">
    <ul>
      <li ng-class="{marcado: nav.estoy('/')}">
        <a href="#/">Datos personales</a>
      </li>
      <li ng-class="{marcado: nav.estoy('/descargas')}">
```

```

        <a href="#/descargas">Descargas</a>
    </li>
    <li ng-class="{marcado: nav.estoy('/opciones')}">
        <a href="#/opciones">Opciones de cuenta</a>
    </li>
</ul>
</nav>
<hr />
<div ng-view></div>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular-route.js"></script>
<script src="app.js"></script>
</body>

```

De momento observarás que en el HTML anterior solo parece que estemos usando un controlador, en la etiqueta NAV. No nos hemos olvidado del otro, debes recordar que estaba declarado para cada una de las vistas, que habíamos definido con el `$routeProvider`.

```

$routeProvider
    .when("/", {
        controller: "appCtrl",
        controllerAs: "vm",
        templateUrl: "home.html"
    })
    // ....

```

Controlador para la barra de navegación

El controlador `navCtrl`, declarado en la etiqueta NAV, tiene un scope conocido como "nav" (`ng-controller="navCtrl as nav"`). En realidad este controlador es muy sencillo, luego veremos su código. De momento lo usamos simplemente para definir si debe aparecer una clase (class de CSS) en el enlace de la sección tenemos abierta en nuestra aplicación. Ya sabes que cuando se trata de asignar o no una clase se utiliza la directiva `ng-class`, en la que usamos un objeto para definir la clase y la expresión que debe de cumplirse para que esté presente en la etiqueta.

```

<li ng-class="{marcado: nav.estoy('/')}">

```

Si te fijas, se trata de una clase llamada "marcado" que estará presente en la etiqueta LI en caso que el método `estoy()` que está dentro del scope "nav" nos devuelva un valor evaluado como verdadero.

```

<li ng-class="{marcado: nav.estoy('/opciones')}">

```

Este otro elemento de la lista tendrá la clase "marcado" en caso que el método `estoy()` de `nav` devuelva `true` cuando le pasamos el parámetro con valor `"/opciones"`.

Ahora veamos la implementación de ese controlador.

```
.controller("navCtrl", function($location){
    var map = this;
    map.estoy = function(ruta){
        return $location.path() == ruta;
    }
})
```

Como puedes comprobar en el controlador se definió el método `estoy()`, pero hay una cosa nueva que te llamará la atención seguramente. Se trata de la variable `$location` que está inyectada en la función que implementa el controlador.

Conociendo \$location

`$location` es un servicio ("service", tal como los llaman en AngularJS) que nos sirve para mantener y trasladar información de la ruta actual del navegador. `$location` implementa una interfaz para el acceso a la propiedad nativa de Javascript `window.location`, y nos sirve para acceder a elementos de la ruta actual, conocer su estado y modificarlo. Por tanto, podemos saber en todo momento qué ruta tenemos y conocer cuando el usuario ha navegado a otro lugar, ya sea con los botones de atrás o adelante o pulsando un enlace.

Pero no solo eso, existe un un enlace entre `$location` y `window.location` en las dos direcciones. Todo cambio que realicemos en `$location` también tendrá una respuesta en lo que se está mostrando en la barra de direcciones del navegador. Así como todo cambio de URL en el navegador tiene un efecto en el service `$location`.

`$location` tiene una serie de métodos bastante interesante a través de los cuales podemos acceder a partes de la URL que está en la barra de direcciones, desde el dominio, el puerto, protocolo, hasta la ruta en sí de la aplicación.

En el ejemplo de antes estábamos usando `$location.path()` que nos devuelve el camino actual. Este método acepta dos juegos de parámetros.

- Si llamamos a `$location.path()` sin parámetros, nos devuelve la ruta actual.
- Si llamamos a `$location.path()` indicando un parámetro, entonces se cambia la ruta actual a aquello que le hayamos indicado. La ruta debe comenzar por "/" el método es lo suficientemente listo para que, si no tiene la barra, se la pone automáticamente.

Así que, volviendo a nuestra función `estoy()`, apreciarás que devolverá `true` cuando la ruta enviada como parámetro coincida con la ruta de nuestra aplicación en ese instante, recuperada con `$location.path()`.

```
map.estoy = function(ruta){
    return $location.path() == ruta;
}
```

Controlador de las vistas

De nuevo llegamos al controlador de las vistas independientes. Ahora hemos colocado un poco de código.

```
controller("appCtrl", function(){
    var vm = this;
    vm.colores = ["green", "blue", "orange"];
});
```

Realmente este código no es más que una inicialización de valores en un array llamado "vm.colores".

Ahora veamos el código de las vistas independientes y veamos cómo usamos ese array de colores. Observarás que las vistas no hace nada muy espectacular, solo hemos colocado alguna cosa con fines didácticos.

Vista "home.html": un simple recorrido a los colores del array para mostrarlos en cajas DIV.

```
<h1>Home</h1>
<div ng-repeat="color in vm.colores">{{ color }}</div>
```

Vista "opciones.html": un campo de texto donde escribir un color nuevo. Un botón donde enviamos el contenido de ese campo de texto dentro del array de colores.

```
<h1>Opciones</h1>
Tienes {{ vm.colores }}
<p>
  Agrega <input type="text" ng-model="vm.nuevocolor"> <input type="button" value="ok" ng-click="vm.colores.push(vm.nuevocolor)">
```

Vista "descargas.html": Otro botón para vaciar el array de colores.

```
<h1>Descargas</h1>
{{ vm.colores }}
<input type="button" value="Vaciar" ng-click="vm.colores = []">
```

Controladores siempre inicializan sus valores en cada "uso"

Ahora queremos mencionar de nuevo la problemática de los controladores comentada por encima en el artículo anterior. Ahora que tenemos algo de código en nuestro controlador de las vistas y somos capaces de ejecutarlo para darnos cuenta de un detalle.

Si realizas una navegación por las distintas secciones de la aplicación fabricada hasta el momento verás que todas las vistas operan de alguna forma sobre el array de los colores. Podemos en una vista agregarle elementos, en otra podemos vaciarlo. Sin embargo, te habrás fijado que cada vez que se accede a una vista el array de colores vuelve a su estado inicial, independientemente que antes lo hayamos manipulado. ¿Cómo es posible? ¿Entonces no hay manera de memorizar los estados del scope entre las diferentes vistas de una aplicación? Como verás más tarde, sí la hay, pero no lo haremos con los controladores.

Este comportamiento ocurre porque, cada vez que se accede a una vista independiente se está volviendo a cargar el controlador, ejecutándose la función del controlador. Como en el controlador se inicializa el array

de colores, observarás que cada vez que se usa en una vista ese array se inicializa de nuevo. En fin, cuando pasas de una vista a otra el array de colores se vuelve a poblar con los colores que se habían configurado de manera predeterminada en el controller.

La solución a este problema pasa por usar factorías o servicios, pero es algo que veremos en el siguiente artículo. Hasta ahora es importante que te quedes con esto: las funciones que implementan los controladores son como constructores, que se llaman cada vez que el compilador del HTML de Angular pasa por la vista.

Este artículo es obra de *Alberto Basalo*

Fue publicado por primera vez en 05/12/2014

Disponible online en <http://desarrolloweb.com/articulos/introduccion-location-primeros-controladores-paralelo-angularjs.html>

Otros artefactos en AngularJS

Vamos repasando otros recursos para la programación de aplicaciones en AngularJS, de diversa índole, como factorías, servicios y analizando en profundidad asuntos importantes como el scope.

Factorías (factory) en AngularJS

Qué son las factorías de AngularJS, factory en la terminología Angular. Para qué sirven, qué rasgos las caracterizan y cómo crearlas.

Hasta el momento en el [Manual de AngularJS](#) nuestro código Javascript lo hemos colocado principalmente dentro de controladores. Nos han servido perfectamente para nuestras aplicaciones, pero ya hemos podido observar que en ciertas ocasiones los controladores no se comportan como nosotros podríamos desear.

Aspectos de esta problemática han debido de quedar claros si te lees con calma el último artículo, donde [aprendimos a crear controladores en paralelo y donde introducimos \\$location](#). En ese ejercicio vimos que los controladores se invocan con cada vista donde los estemos usando, ejecutando la función que los define cada vez que se carga la vista. Por ese motivo todos los datos que se se inicializan en los controladores se vuelven a poner a sus valores predeterminados cuando cargo cualquier vista que trabaje con ese controlador.



Terminamos el artículo pasado mencionando que las factorías nos podrían solucionar la pérdida de datos de los controladores cuando cambiamos la vista. De modo que vamos a aprender a crearlas y utilizarlas.

Pero ese no es el único caso donde encontrarás utilidad en las factorías. Por ejemplo, algunas de las necesidades que podríamos tener y que los controladores no nos resuelven son:

1. Compartir datos entre varios controladores, lo que permite tener aplicaciones de verdad, capaces de memorizar estados entre varias de sus pantallas.
2. Compartir datos entre varias vistas distintas. Por supuesto, sin usar las temidas variables globales. Eso es justamente lo que vimos en el artículo anterior, que ya tengas uno o varios controladores, no comparten ni se memorizan estados al pasar de una vista a otra.

3. Empaquetar operaciones comunes a varios controladores (por ejemplo en una aplicación de facturación podríamos necesitar calcular el IVA o acceder a los distintos tipos de IVA en varios puntos del sistema). Por supuesto, no queremos colocar el código de esos cálculos u operaciones repetido en todos los controladores que deben utilizarlos y tampoco deseamos crear funciones con ámbito global.

Qué son las factorías

Las factorías son como contenedores de código que podemos usar en nuestros sitios desarrollados con AngularJS. Son un tipo de servicio, "service" en Angular, con el que podemos implementar librerías de funciones o almacenar datos.

Cuando las usamos tienen la particularidad de devolvernos un dato, de cualquier tipo. Lo común es que nos devuelvan un objeto de Javascript donde podremos encontrar datos (propiedades) y operaciones (métodos). Con diferencia de los controladores, las factorías tienen la característica de ser instanciados una única vez dentro de las aplicaciones, por lo que no pierden su estado. Por tanto, son un buen candidato para almacenar datos en nuestra aplicación que queramos usar a lo largo de varios controladores, sin que se inicialicen de nuevo cuando se cambia de vista.

Angular consigue ese comportamiento usando el patrón "Singleton" que básicamente quiere decir que, cada vez que se necesite un objeto de ese tipo, se enviará la misma instancia de ese objeto en lugar de volver a instanciar un ejemplar.

Nota: El patrón "Singleton" no es algo específico de AngularJS, en realidad es un patrón general de programación orientada a objetos. Así como las factorías en líneas generales también son un conocido patrón de diseño de software que se usa en el desarrollo de aplicaciones web y aplicaciones tradicionales orientadas a objetos.

Notas sobre los "services" en Angular

Los "services" en AngularJS incluyen tanto factorías como servicios. Más adelante veremos la diferencia. Lo que queremos mencionar ahora es que estos contenedores de código ya los hemos usado en diversas ocasiones y quizás podrás entender su utilidad mejor si analizamos cosas que ya conoces.

Por ejemplo, cuando estás haciendo Ajax, por los métodos que hemos conocido hasta el momento, usamos \$http. Éste no es más que un service de Angular que engloba toda la funcionalidad necesaria para realizar solicitudes asíncronas a un servidor.

Por tanto, algo como Ajax, que se supone que puedes querer realizar a lo largo de tu aplicación en varias partes del código, se ha separado a un "servicio". Esto quiere decir que, cuando quieras hacer Ajax, tendrás que usar el código del "service" \$http.

Los servicios y factorías que desees usar en tus controladores o módulos deberás inyectarlos como has visto hacer en diversas partes de nuestros ejemplos. No te preocupes si no lo recuerdas porque a continuación veremos ejemplos.

Ejemplo de factoría en AngularJS

Ahora nos vamos a poner manos a la obra creando nuestra primera factoría. Para ello vamos a continuar con el ejercicio que hemos visto en los artículos anteriores del Manual de AngularJS. Como recordarás, queremos implementar un sistema que nos memorice cierta información de nuestra aplicación a lo largo de diversas vistas.

Implementamos factorías con el método `factory()` que depende del módulo (objeto `module`) de nuestra aplicación.

Osea, como cualquier aplicación de Angular, lo primero crearás tu "module" principal:

```
angular.module("app", ["ngRoute"])
```

Y sobre el objeto que devuelve esa operación crearemos las factorías.

Nota: Como observarás, el mecanismo para crear la factoría es el mismo que hacemos para crear los controladores. Para crear el controlador usas el método `controller()` y para la factoría el método `factory()`.

```
.factory("descargasFactory", function(){
    var descargasRealizadas = ["Manual de Javascript", "Manual de jQuery", "Manual de AngularJS"];

    var interfaz = {
        nombre: "Manolo",
        getDescargas: function(){
            return descargasRealizadas;
        },
        nuevaDescarga: function(descarga){
            descargasRealizadas.push(descarga);
        }
    }
    return interfaz;
})
```

Esta factoría se llama "descargasFactory". El nombre lo hemos definido en la llamada al método `factory`. Acuérdate de este nombre, pues luego lo tendrás que usar al inyectar la dependencia de esta factoría en tus controladores.

Ese código tiene una serie de detalles interesantes, desde el punto de vista de Angular y también desde el de Javascript en general. Estudiar el código anterior con detalle es suficiente para un único artículo, porque entran en juego diversos conceptos de la programación orientada a objetos en Javascript. De todos modos, te vamos a resumir un poco lo que encuentras.

- Lo más destacado sobre las factorías en AngularJS lo encuentras en la última línea: "return interfaz;"

Todas las factorías deben devolver algo. Lo que sea, aunque lo habitual como dijimos es devolver un objeto. Por definición debe de ser así en AngularJS.

- Aquello que devuelves es lo que se conoce desde fuera de la factoría. Por decirlo de otra manera, es la interfaz pública de uso de esa factoría. Por eso hemos llamado a la variable que devolvemos en el return "interfaz", porque es la serie de propiedades y métodos que estás haciendo público para todo el mundo que use esa factoría. Lógicamente, esa "interfaz" no es más que una manera nuestra de llamar a la variable y tú usarás la que quieras.
- Pero fíjate que la variable "descargasRealizadas" es privada a la factoría, pues no se devuelve en la interfaz. Por tanto, ese array no podrá ser accesible desde fuera de la factoría. Podemos entenderlo como una propiedad privada.
- Para acceder al array "descargasRealizadas" se hará uso de los métodos definidos en "interfaz": getDescargas() y nuevaDescarga(). Esos métodos son públicos, por haberlos definido en la interfaz que devolvemos en la función de la factoría y se podrán acceder desde cualquier lugar donde tengamos disponible la factoría.
- Sin embargo no todos los datos que vamos a manejar en las factorías necesitamos hacerlos privados. En concreto encontrarás, la propiedad "nombre" que está dentro de nuestra interfaz y por lo tanto es pública y podrá accederse tal cual desde fuera de la factoría.

Nota: Para entender las ventajas hacer las cosas públicas o privadas deberás conocer algunos de los conceptos básicos de programación en general y programación orientada a objetos en particular, como son la abstracción y el encapsulamiento. Es una decisión de diseño de software, que debe tomar el desarrollador y tanto Javascript como por extensión AngularJS tienen mecanismos para implementar elementos públicos o privados. Así, como regla global en la programación orientada a objetos, todo dato debería ser definido como privado, a no ser que alguien de fuera te solicite que lo expongas públicamente.

Usar una factoría

Ahora podemos ver cómo usar la factoría que acabamos de realizar. El procedimiento es tan simple como, una vez definida, inyectarla en el controlador donde la queremos usar. Usamos el sistema de inyección de dependencias que ya conoces.

Al crear la función del controlador debemos definir un parámetro que se llame exactamente igual al nombre que le has dado en la factoría. En este caso el parámetro que inyectamos en el controlador se llama "descargasFactory" pues así habíamos llamado a la factoría al crearla.

Echa ahora un vistazo a un controlador que usa esta factoría.

```
.controller("appCtrl", function(descargasFactory){
    var vm = this;

    vm.nombre = descargasFactory.nombre;
    vm.descargas = descargasFactory.getDescargas();
    vm.funciones = {
        guardarNombre: function(){
            descargasFactory.nombre = vm.nombre;
        },
    },
}
```

```
agregarDescarga: function(){
    descargasFactory.nuevaDescarga(vm.nombreNuevaDescarga);
    vm.mensaje = "Descarga agregada";
},
borrarMensaje: function(){
    vm.mensaje = "";
}
}
});
```

Dentro de nuestro controlador la variable `descargasFactory` que recibimos como parámetro contiene todos los datos y funciones que hemos definido en la interfaz pública de nuestra factoría.

Por tanto:

- `descargasFactory.nombre` contendrá la propiedad "nombre" definida en la factory.
- `descargasFactory.nuevaDescarga()` o `descargasFactory.getDescargas()` serán llamadas a los métodos que habíamos definido en la factoría.

Nota: Ojo, desde tu vista, en el HTML, no serás capaz de acceder a la factoría. Es algo que ahora pertenece al controlador y a través del scope creado por ese controlador no puedes acceder a los datos que tienen sus dependencias. Si quieres acceder a un dato de esa factoría el mecanismo es volcar ese dato en el scope. Eso lo hacemos al principio de ese controlador en las líneas:

```
vm.nombre = descargasFactory.nombre;
vm.descargas = descargasFactory.getDescargas();
```

Creo que con todo esto queda explicado el trabajo con las factorías en AngularJS. Ahora faltaría un poco de tiempo por tu parte para poder ponerlo en práctica.

Este artículo es obra de *Alberto Basalo*

Fue publicado por primera vez en 15/12/2014

Disponible online en <http://desarrolloweb.com/articulos/factorias-factory-angularjs.html>

Scope en Angular, manejando ámbitos con \$parent

Explicaciones más detalladas del omnipresente scope de AngularJS, nuestro ámbito. Introducimos el `$parent` para acceder al scope padre.

El scope, un término que encontrarás nombrado hasta la saciedad en la literatura relacionada con este framework de Javascript. ¿Qué es exactamente el scope? ¿Qué podemos decir sobre él para completar lo visto hasta ahora en el [Manual de AngularJS](#)? Sobre todo ello vamos a hablar en este artículo.

La traducción de scope es "ámbito", sin embargo, el término es tan habitual que solemos usar directamente la palabra en inglés. Pero creo que usando la traducción nos podemos enterar mejor qué es en realidad el scope: el "ámbito" de los datos donde estamos trabajando en las vistas.



En Angular no existe un único ámbito, puedes tener (tendrás) diferentes scopes en diferentes partes de tu código, anidados o en paralelo. Los scope se crean automáticamente para cada uno de tus controladores y además existe un scope raíz para toda tu aplicación. A continuación te enseñaremos a trabajar con ellos, almacenando datos y recuperándolos.

Scope hace de enlace entre las vistas y los controladores

El scope es la magia que permite que los datos que manejas en los controladores pasen a las vistas, y viceversa. Es el enlace que nos traslada esos datos de un lugar a otro, sin que nosotros tengamos que programar nada.

En los controladores hemos aprendido a generar elementos en el scope de diversas maneras. Lo común es acceder al scope si lo inyectas en el controlador mediante el parámetro `$scope`. Luego podrás añadir datos simplemente como propiedades a ese objeto `$scope` inyectado:

```
.controller('otroCtrl', function($scope){  
  $scope.algo = "probando scope...";  
  $scope.miDato = "otra cosa..."  
});
```

Eso te generará un dato en tu scope llamado "algo" y otro llamado "miDato".

Desde las vistas puedes acceder a datos de tu scope a través de expresiones del tipo:

```
{{ algo }}
```

Ese "algo" es un dato que tienes almacenado en el scope desde el controlador. Como sabes, con esa expresión lo vuelcas en la página. También podrás acceder al scope cuando usas la directiva `ng-model` y defines un dato en tu modelo.

```
<input type="text" ng-model="miDato">
```

Como se decía, el scope nos sirve para trasladar esos datos entre el controlador y la vista. Nosotros para poder acceder a "algo" o a "miDato" no necesitamos enviarlo desde el controlador a la vista ni de la vista al controlador, salvo crear esa propiedad en el `$scope`. Lo bonito, que vimos en el artículo de [Binding en Angular](#), es que cuando el dato se modifica, ya sea porque se le asigna un nuevo valor desde el controlador o

porque se cambia lo que hay escrito en el INPUT de la vista, la otra parte es consciente de ese cambio sin tener que suscribirnos a eventos.

Ámbitos del scope dentro de la vista

Si lees eso en español "ámbito del scope" parece un trabalenguas o quizás una frase infantil, "ámbito del ámbito". Queremos decir que cada scope tiene su ámbito restringido a una parte del código, aunque la propia palabra "scope" ya nos dice por ella misma que se trata de eso.

Bueno en definitiva, vamos a echar un vistazo a un HTML:

```
<div ng-controller="miAppController">
  <p>Contenido de nuestra vista acotado por un controlador</p>
  <p>{{ cualquierCosa }}</p>
</div>
<section>
  Desde fuera de esa división no tengo acceso al scope del controlador
</section>
```

Lo que queremos que se vea ahora es que el scope creado por ese controlador "miAppController" tiene validez dentro de ese código HTML en el lugar donde se ha definido la directiva ngController. Es decir, los elementos que asignes a \$scope dentro de tu controlador se podrán ver desde la etiqueta DIV y todas sus hijas, pero no desde etiquetas que se encuentren fuera, como es el caso del SECTION que hay después.

Alias del scope

Existe otra modalidad de enviar datos al scope y es por medio de un alias. Esto te sirve cuando al declarar tus controladores en el HTML (directiva ng-controller) usas "controller..as".

```
<div ng-controller="pruebaAppCtrl as vm">
```

En este caso podrás generar datos en el scope de dos maneras, o bien a través del mismo objeto \$scope que puedes inyectar, tal como acabamos de explicar, o bien a través de la variable this dentro de la función.

```
.controller('pruebaAppCtrl', function($rootScope){
  var modeloDeLaVista = this;
  modeloDeLaVista.otroDato = "Esto está ahora en el scope, para acceder a través de un alias";
});
```

Gracias al alias en la vista podrás acceder a ese dato con una expresión como esta:

```
{{ vm.otroDato }}
```

Este alias del scope nos simplificará la vida cuando estemos trabajando con varios ámbitos. Enseguida lo veremos.

Ámbito raíz con \$rootScope

En AngularJS existe un ámbito o scope raíz, sobre el que podemos insertar datos. Ese ámbito es accesible desde cualquier lugar de la aplicación y nos puede servir para definir valores que puedes acceder desde cualquier punto de tu HTML, independientemente del controlador donde te encuentres. El scope raíz de Angular lo puedes acceder desde tu controlador, por medio de \$rootScope, que necesitas inyectar en la función de tu controlador en caso que desees usarlo.

```
.controller('pruebaAppCtrl', function($scope, $rootScope){  
    $scope.scopeNormal = "Esto lo coloco en el scope normal de este controlador...";  
    $rootScope.scopeRaiz = "Esto está en el scope raíz";  
});
```

Cuando guardas algo en el ámbito raíz puedes acceder a ese valor, desde tu vista, a través del nombre de la propiedad que has creado en \$rootScope.

```
<div ng-controller="pruebaAppCtrl">  
    {{ scopeNormal }}  
    <br />  
    {{ scopeRaiz }}  
</div>
```

Como ves, en este ejemplo accedemos de la misma manera a un dato que tienes en el scope de este controlador y a un dato que tiene en el scope raíz. Esto es así porque en AngularJS, si una variable del modelo no se encuentra en el scope actual, el propio sistema busca el dato en el scope padre. Si no, en el padre y así sucesivamente.

Conociendo \$parent

El problema es cuando tienes un dato con el mismo nombre en dos scopes distintos.

```
.controller('pruebaAppCtrl', function($scope, $rootScope){  
    //sobrescribo una variable del scope  
    //en realidad son dos datos distintos con dos valores distintos  
    //uno lo tengo en el scope del controlador y otro en el scope raíz "root"  
    $scope.repetido = "Algo en el scope normal";  
    $rootScope.repetido = "Algo en el scope raíz";  
});
```

En ese caso, si en la vista hacemos algo como {{ repetido }} verás que existe una ambigüedad, pues ese valor puede significar dos cosas, dependiendo si miramos en un scope u otro. Angular resuelve esa situación devolviendo el dato del scope más específico. Osea, te dará el valor que tiene en \$scope.repetido y no podrás acceder a \$rootScope.repetido, a no ser que uses \$parent.

\$parent permite acceder al scope "padre", eso en caso que tengas dos controladores uno dentro de otro. O en el caso que solo tengas un controlador con \$parent podrás acceder al scope raíz.

Así pues, para acceder a ambos valores podría usar un HTML como este:

```
<div ng-controller="pruebaAppCtrl">
  {{ repetido }} --- {{ $parent.repetido }}
</div>
```

El primer "repetido" nos permite acceder al dato que tenemos en el scope actual y por su parte \$parent.repetido nos permite acceder al dato que habíamos guardado sobre \$rootScope.

Ejemplo con ámbitos corrientes y ámbito root

Todo el código que hemos visto desmembrado en este artículo se puede resumir en el siguiente ejemplo.

HTML

```
<div ng-app="pruebaApp" ng-controller="pruebaAppCtrl">
  {{ scopeNormal }}
<br />
  {{ scopeRaiz }}
<br />
  {{ algo }} --- {{ $parent.algo }}
</div>
```

Javascript:

```
.controller('pruebaAppCtrl', function($scope, $rootScope){
  $scope.scopeNormal = "Esto lo coloco en el scope normal de este controlador...";
  $rootScope.scopeRaiz = "Esto está en el scope raíz";

  //sobrescribo una variable del scope
  $scope.algo = "Algo en el scope normal";
  $rootScope.algo = "Algo en el scope raíz";
});
```

Puedes apreciar más o menos todo lo que venimos comentando sobre los ámbitos, pero lo verás mejor de una manera visual con la extensión Angular Batarang.

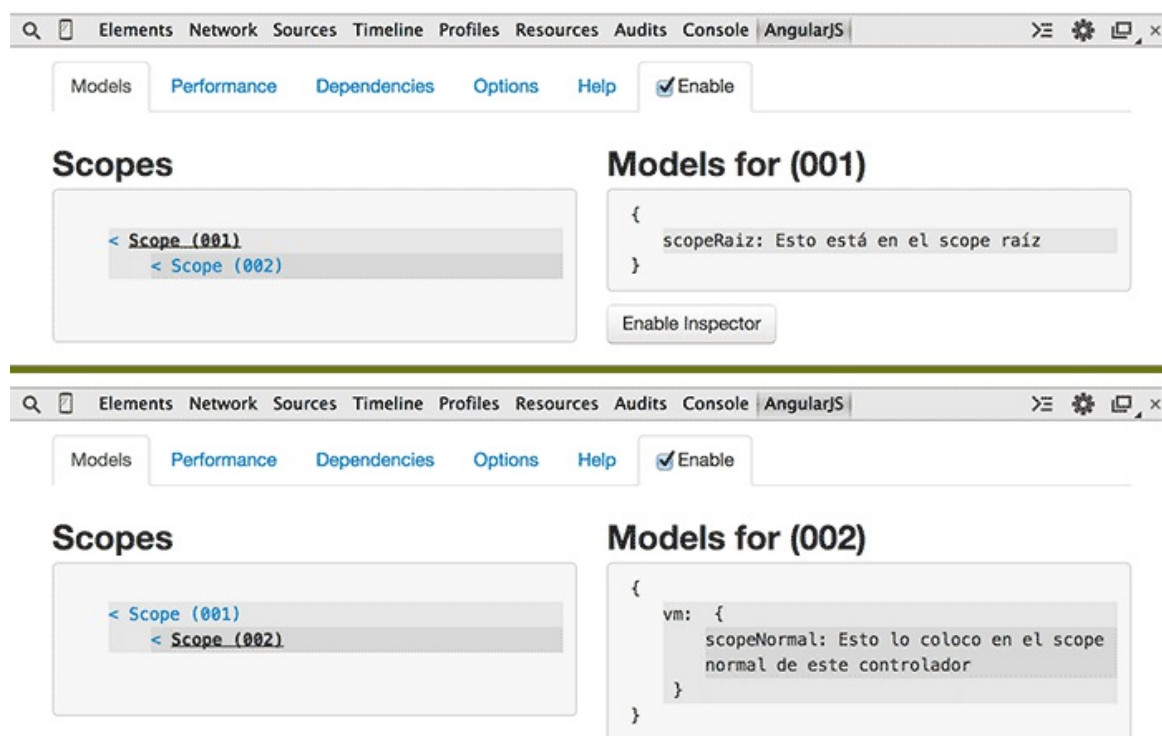
Angular Batarang

Existe una herramienta que se integra en Google Chrome que te sirve de inspector de scopes en AngularJS. Se llama "Batarang" y nos permite ver de una manera rápida lo que tenemos en los diferentes scopes de nuestra aplicación.

Para desarrollar con AngularJS es imprescindible. Seguro que la conoces.

Si inspeccionamos el ejercicio anterior con la extensión "AngularJS Batarang" podrás encontrar más o

menos lo que ves en esta imagen:



Como ves, las herramientas para desarrolladores de Chrome tienen ahora una nueva pestaña que me permite examinar los distintos scopes de esta aplicación. En la imagen anterior puedes ver dos pantallas, examinando el scope raíz y el scope común generado por un controlador.

Finalizamos por aquí, a la espera de publicar un artículo que nos permita construir controladores anidados, o en paralelo, donde podremos observar también diferentes construcciones de ámbitos.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 22/12/2014
Disponible online en <http://desarrolloweb.com/articulos/scope-angular-parent.html>

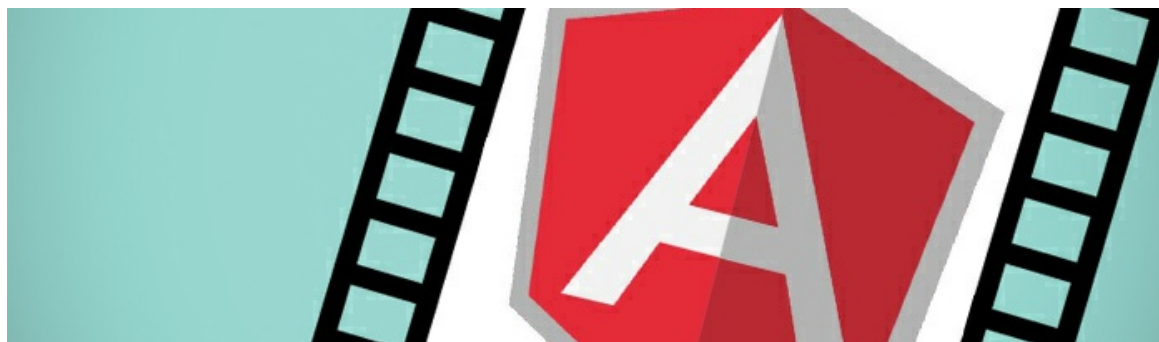
Ámbito o scope en controladores anidados y paralelos

Realizamos diversas variantes de construcción de aplicaciones en Angular con distintos controladores y repasamos las condiciones del scope en cada una.

En el artículo anterior pudimos aprender bastantes detalles sobre el "famoso" scope de AngularJS. Vimos que se van creando a medida que se usan los controladores dentro de la página y conocimos que existen diversos ámbitos o distintos "scopes" en una página. Ahora vamos a ver que cuando se trabaja con varios controladores se producen varios ámbitos diferentes y que se puede acceder a unos u otros dependiendo las necesidades y situaciones.

Cuando se trata de controladores, los podemos definir "en paralelo" y "anidados" en cada situación los scopes también se colocarán lado a lado o bien unos dentro de otros. Lo veremos mejor con ejemplos de

código.



Controladores en paralelo

En este caso tenemos un controlador al lado del otro, en el código HTML. No es la primera vez en este Manual de AngularJS que implementamos controladores en paralelo, así que no debería de representar mucho problema.

Para hacer dos controladores funcionando en paralelo podremos tener un HTML en el que se define un controlador al lado del otro.

```
<body ng-app="app">
  <header ng-controller="headerCtrl">
    {{ scopeHeader }}
  </header>
  <section ng-controller="sectionCtrl">
    {{ scopeSection }}
  </section>
</body>
```

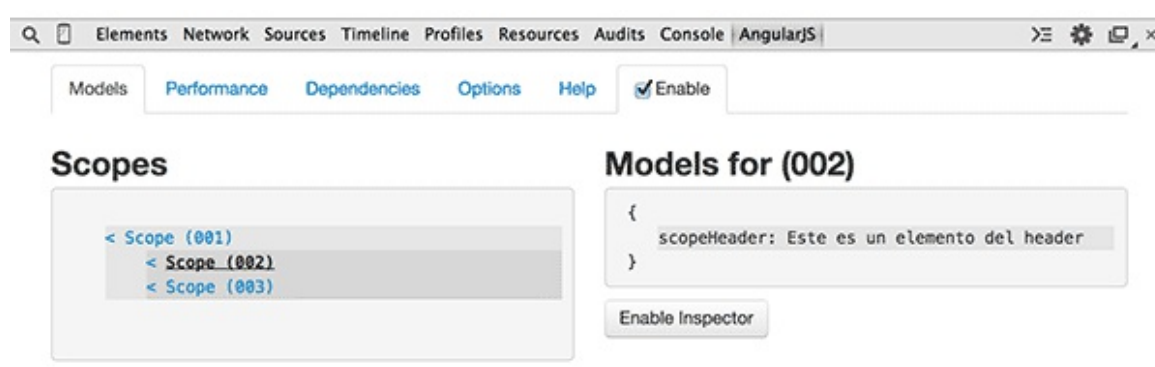
El Javascript con el que se declaran y definen estos controladores lo puedes ver a continuación:

```
var app = angular.module("app", [])

var headerController = function($scope){
  $scope.scopeHeader = "Este es un elemento del header";
};
app.controller("headerCtrl", headerController);

var sectionController = function($scope){
  $scope.scopeSection = "Este es un elemento del section";
};
app.controller("sectionCtrl", sectionController);
```

Como sabes, en cada controlador se crea un scope son independiente. Todos dependen de un scope raíz que sigue existiendo y dentro de éste tendremos los dos scopes de cada controlador. Si inspeccionamos estos ámbitos con la extensión Batarang obtendremos esto:



Desde un scope no puedo acceder al otro. Eso quiere decir que los elementos definidos para el HEADER, como la propiedad "\$scope.scopeHeader", no se puede acceder desde el controlador definido en el SECTION.

Nota: En general, si quisiéramos compartir datos entre ambos controladores nos tocaría implementar una factoría o servicio, o quizás si lo consideramos más adecuado para nuestro caso, crear datos en el scope raíz, inyectando \$rootScope en los controladores y asignando propiedades en ese objeto. El \$rootScope lo conocimos en el artículo titulado [Scope en Angular, manejando ámbitos con \\$parent](#).

Controladores anidados

Las características y las posibilidades de navegación entre distintos scopes las vemos mejor todavía en el caso de los controladores anidados. En este caso tenemos un HTML en el que se coloca un controller y dentro de éste otro pedazo de HTML donde se declara otro elemento con un nuevo controller.

```
<body ng-app="app">
  <section ng-controller="sectionCtrl">
    <p>
      {{ scopeSection }}
    </p>
    <article ng-controller="articleCtrl">
      {{ scopeArticle }}
    </article>
  </section>
</body>
```

Ahora, en la declaración de los controllers en Javascript, observarás que no hay ningún indicio que en el HTML estén anidados. De hecho es lo correcto, pues los controladores deben ser agnósticos a cómo está hecho nuestro HTML.

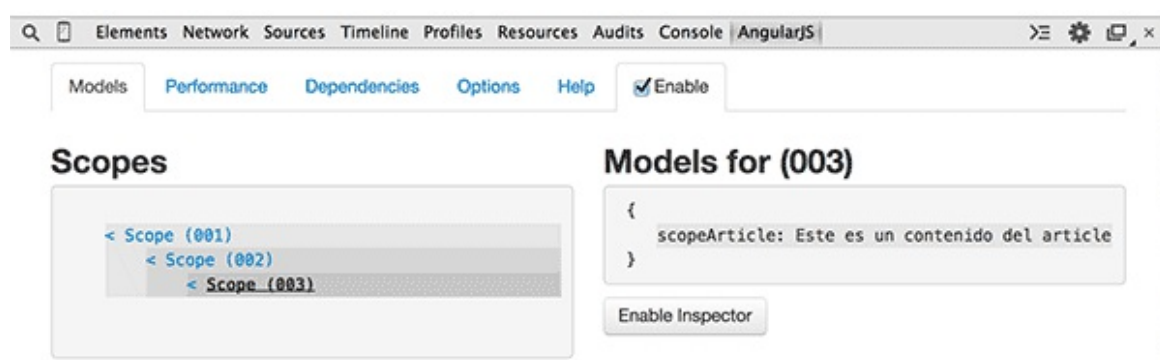
```
var app = angular.module("app", [])

var sectionController = function($scope){
  $scope.scopeSection = "Este es un elemento del section";
};

app.controller("sectionCtrl", sectionController);
```

```
var articleController = function($scope){  
    $scope.scopeArticle = "Este es un contenido del article";  
};  
app.controller("articleCtrl", articleController);
```

En este caso podremos observar que los ámbitos están uno dentro de otro, gracias a la extensión Batarang.



Lo interesante en este caso es que podemos acceder desde un scope a los datos declarados en un scope padre, gracias a la anidación definida en el HTML.

```
<section ng-controller="sectionCtrl">  
  <p>  
    {{ scopeSection }}  
  </p>  
  <article ng-controller="articleCtrl">  
    {{ scopeArticle }}  
    <p>  
      Si quiero, puedo acceder al scope del elemento padre <b>{{ scopeSection }}</b>  
    </p>  
  </article>  
</section>
```

Como puedes ver, dentro del ARTICLE, definido con controller "articleCtrl" podemos acceder perfectamente a un dato que está en el scope del controlador padre: {{scopeSection}}.

Nota: Esto, que ya se explicó, es gracias a que en Angular, si no se encuentran los datos en el scope actual, automáticamente se busca en el scope padre, subiendo todos los niveles que haga falta hasta llegar al scope raíz. Ese es un comportamiento automático para el cual no necesitamos configurar ni programar nada.

Uso de \$parent para ir al scope padre

También explicamos ya que es posible usar \$parent dentro de nuestra vista, para acceder al scope padre. Pero ahora lo vamos a ver con controladores anidados.

La variable `$parent` contiene simplemente una referencia al scope padre. Éste puede ser el scope raíz o si tenemos controladores anidados es el del nivel superior en la anidación. El código HTML anterior podríamos haberlo escrito así.

```
<section ng-controller="sectionCtrl">
  <p>
    {{ scopeSection }}
  </p>
  <article ng-controller="articleCtrl">
    {{ scopeArticle }}
    <p>
      Si quiero, puedo acceder al scope del elemento padre <b>{{ $parent.scopeSection }}</b>
    </p>
  </article>
</section>
```

Prácticamente no hay ninguna diferencia, excepto que cuando accedemos desde el ARTICLE a un modelo (dato) del SECTION lo hacemos a través de `$parent`:

```
{{ $parent.scopeSection }}
```

En este caso usando `$parent` el código puede quedar un poco más claro pues el que lo lea sabrá que ese modelo está en el scope del controlador padre. Sin embargo, enseguida veremos otra manera de hacer esto en la que el código puede quedar más claro todavía.

Sobrescribir variables del modelo en scopes anidados

La situación que nos puede surgir en nuestras aplicaciones es que tengamos una misma propiedad en un scope y en un scope anidado. En una circunstancia como esa, estaremos obligados a usar `$parent` para acceder al modelo del scope padre. Esto lo veremos mejor con un ejemplo.

```
<section ng-controller="manualCtrl">
  <p>
    Manual: {{ nombre }}
  </p>
  <article ng-controller="articuloCtrl">
    Artículo: {{ nombre }}
    <p>
      Este artículo pertenece al manual {{ $parent.nombre }}
    </p>
  </article>
</section>
```

Como puedes observar, tanto el "manualCtrl" como "articuloCtrl" tienen un modelo llamado "nombre" que será el título del manual o el título del artículo. Si desde el controller del artículo queremos acceder al nombre del manual, no podemos escribir `{{ nombre }}` porque nos saldría el título del artículo. En ese caso tenemos que escribir la ruta `$parent.nombre`.

Esta situación y uso de `$parent` se puede extender a múltiples niveles de anidación. Por ejemplo podríamos tener comentarios dentro de los artículos y en ellos el modelo "nombre" que en este caso sería el autor del comentario. Desde el scope de los comentarios podríamos acceder a los dos scope padre, tanto artículo como el manual, encadenando `$parent`.

```
<section ng-controller="manualCtrl">
  <p>
    Manual: {{ nombre }}
  </p>
  <article ng-controller="articuloCtrl">
    Artículo: {{ nombre }}
    <p>
      Este artículo pertenece al manual {{ $parent.nombre }}
    </p>
    <p ng-controller="comentariosCtrl">
      Este comentario lo escribe {{ nombre }}
      <br>
      Pertenece al artículo {{ $parent.nombre }}
      <br>
      Que a su vez pertenece al manual {{ $parent.$parent.nombre }}
    </p>
  </article>
</section>
```

Sintaxis con controller..as

Ahora, creo que todos estaremos de acuerdo en que algo como `{{ $parent.$parent.nombre }}` no ayuda a la legibilidad del código. Es cierto que sabemos con leer esa expresión que estamos accediendo a un ámbito padre, en este caso dos veces seguidas, pues será el padre dos niveles por arriba. Pero esa complejidad de anidación de los controladores puede que no esté en nuestra cabeza y ya no nos acordemos que hay tantos niveles para arriba.

En fin, que existe una sintaxis que nos mejora el uso que hemos resumido de `$parent` a través de distintos espacios de nombres. No es otra solución que "controller..as" que ya debes de conocer.

Nota: La sintaxis del "controller as" la hemos usado en diversas ocasiones a lo largo de este manual. La explicamos por primera vez en el artículo sobre los [códigos para hacer un controlador y sus variantes](#).

```
<section ng-controller="manualCtrl as manual">
  <p>
    Manual: {{ manual.nombre }}
  </p>
  <article ng-controller="articuloCtrl as articulo">
    Artículo: {{ articulo.nombre }}
    <p>
      Este artículo pertenece al manual {{ manual.nombre }}
    </p>
  </article>
</section>
```

```
<p ng-controller="comentariosCtrl as comentario">
  Este comentario lo escribe {{ nombre }}
<br>
  Pertenece al artículo {{ articulo.nombre }}
<br>
  Que a su vez pertenece al manual {{ manual.nombre }}
</p>
</article>
</section>
```

Supongo que se aprecian las diferencias. Ahora cada uno de los scopes que tenemos en cada controlador tiene un alias y dentro de ese controller y todos los hijos existe ese alias. A través del alias somos capaces de acceder al modelo declarado en cada controlador, no solo en el que estamos, sino también en todos los controladores padre.

Este modo de usar los controladores produce un cambio de operativa en nuestro Javascript que ya se explicó también. Dejamos el código fuente para que encuentres las diferencias.

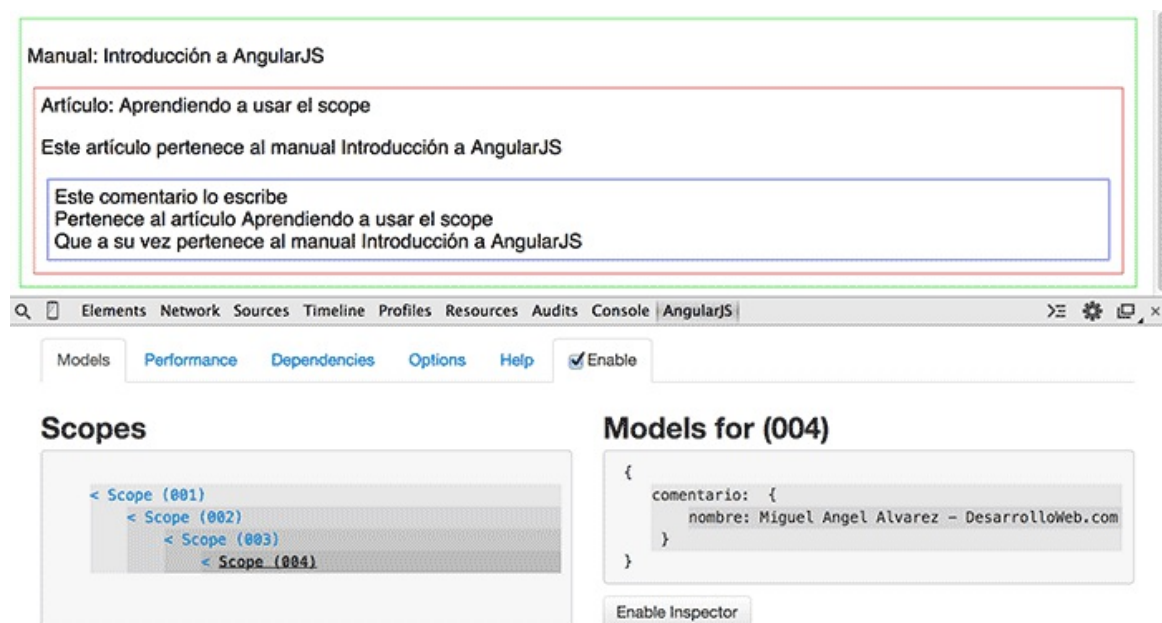
```
var app = angular.module("app", [])

var manualCtrl = function(){
  var vm = this;
  vm.nombre = "Introducción a AngularJS";
};
app.controller("manualCtrl", manualCtrl);

var articuloCtrl = function(){
  var vm = this;
  vm.nombre = "Aprendiendo a usar el scope";
};
app.controller("articuloCtrl", articuloCtrl);

var comentariosCtrl = function(){
  var vm = this;
  vm.nombre = "Miguel Angel Alvarez - DesarrolloWeb.com";
};
app.controller("comentariosCtrl", comentariosCtrl);
```

Puedes ver el aspecto que tendría este ejemplo con el inspector de los scopes de Batarang:



Con esto termina nuestro recorrido a los scopes de AngularJS, esperamos que esta información te haya resultado de utilidad y comprendas mejor el alcance de los ámbitos en los controladores y el modo de trabajar con ellos.

Este artículo es obra de *Miguel Angel Alvarez*.
 Fue publicado por primera vez en 30/12/2014
 Disponible online en <http://desarrolloweb.com/articulos/ambito-scope-controladores-ap.html>

Angular Material

Este módulo para AngularJS nos permite implementar de una manera muy cómoda un diseño de aplicación con las directrices de Material Design de Google.

En este artículo vamos a hacer una review de un proyecto que sin duda resultará muy atractivo para todos los desarrolladores de AngularJS. Se trata de Angular Material, un módulo para AngularJS que nos permite implementar diseños basados en Material Design.

Básicamente lo que nos ofrece no es solo una hoja de estilos, capaz de aplicar la apariencia definida en Material Design, sino toda una serie de componentes que enriquecerán mucho el catálogo de interfaces de usuario disponibles al implementar una web.

Para quien ya viene trabajando desde hace tiempo con AngularJS, podemos hacer un paralelismo con otra librería interesante, llamada [AngularUI](#), que nos ofrece una cantidad enorme de componentes como selectores de fecha, regillas, deslizadores, etc. En este caso Angular Material sería un competidor. Podríamos usarlos juntos, pero cada uno tiene una base de CSS y un look & feel diferente, por lo que quizás no sea tan buena idea.



Angular Material está realizado por Google y puedes acceder al sitio web en la URL:

<https://material.angularjs.org>

Te recomendamos echar un vistazo a los demos, donde encontrarás selectores de fecha, cajas de diálogo, regillas, tarjetas, botones, iconos, barras de tareas, etc.

Angular Material es una excelente opción para programadores, que no tienen habitualmente habilidades de diseño. Usando las directivas que nos ofrece podremos producir aplicaciones que son estéticamente brillantes, junto con una serie de componentes adicionales al propio HTML con un look & feel muy trabajado. Los podremos producir prácticamente sin necesidad de escribir nada de CSS, consiguiendo diseños muy atractivos, aprovechando, pero sin necesidad de dominar, características modernas de CSS como por ejemplo Flexbox.

El único "pero" podríamos decir que es que las aplicaciones se pueden parecer unas a las otras. Para una "home page" de una empresa quizás no es la mejor opción, porque generalmente se deseará una estética diferenciada, pero en el caso de aplicaciones de gestión y aplicaciones web en general es algo interesante, porque nos asegura que los usuarios se sentirán familiarizados con la aplicación. Además, las opciones de personalización también dependerán de nosotros y de cómo queramos modificar el diseño y crear nuestras propias interfaces y componentes, puesto que usar Material Design no implica que estemos restringidos a las directivas que nos dan ya listas.

Qué es Material Design

Material Design es un conjunto de especificaciones sobre un estilo de diseño muy popular. Si tienes un teléfono Android, o accedes a servicios de Google como G+, Inbox, etc. ya sabrás lo que es, aunque quizás no le has dado todavía nombre.

Básicamente lo que nos dicen es cómo podemos diseñar interfaces de usuario, cuáles son las cosas que se deben y no se deben hacer, qué tipos de componentes de interfaces se pueden usar, etc. Como en este artículo no discutimos Material Design os dejamos simplemente un enlace para que consultéis las propias [guías de Google de Material Design](#).

Comenzar con Angular Material

Cuando queremos implementar Material Design en una aplicación AngularJS debemos empezar por descargar los paquetes necesarios para funcionar. Esto lo haremos con nuestro gestor de paquetes de preferencia, ya sea Bower, npm o jspm. Para instalar la última versión estable ejecutaremos los comandos:

Bower:

```
bower install angular-material
```

npm:

```
npm install angular-material
```

jspm:

```
jspm install angular-material
```

A continuación debemos incluir el código de Angular Material en nuestro proyecto, para lo que realizarás los pasos:

Instalación del CSS

Observarás que en los archivos que se descargan hay un código CSS llamado `angular-material.css`. Ese lo tienes que incluir en la cabecera de tu página, con la correspondiente etiqueta:

```
<link rel="stylesheet" href="/ruta_a/angular-material/angular-material.css">
```

Nota: Ojo, que la ruta al archivo dependerá de qué gestor de dependencias estés usando en tu proyecto.

Instalación de los scripts Javascript

Angular Material se instala mediante un script llamado `angular-material.js`, pero antes debemos de instalar también sus dependencias, que son dos módulos de AngularJS que deben ser incluidos también como scripts aparte. Quizás los conozcas ya, son `"angular-aria"` y `"angular-animate"`, el primero se ocupa de temas de accesibilidad y el segundo se usa para conseguir efectos de animación.

Si estás trabajando a la vieja usanza, colocando las etiquetas `SCRIPT` para incluir los js necesarios para tu aplicación, tendrás que incluirlos después de incluir el propio script de AngularJS.

```
<script src="/ruta_a/angular/angular.js"></script>
<script src="/ruta_a/angular-aria/angular-aria.js"></script>
<script src="/ruta_a/angular-animate/angular-animate.js"></script>
<script src="/ruta_a/angular-material/angular-material.js"></script>
```

Nota: De nuevo, las rutas dependerán de si lo has instalado con Bower, npm o jspm.

Declarar las dependencias en el inicio de tu aplicación

El último paso sería indicar en tu aplicación, cuando estás generando tu módulo principal, que vas a usar Angular Material. Seguramente estés familiarizado también con este paso:

```
angular.module( 'YourApp', [ 'ngMaterial' ] )
```

Como puedes observar, de las dependencias de Angular Material no te tienes que preocupar, puesto que el módulo ngMaterial ya se encarga de incluirlas, simplemente te tienes que asegurar que los archivos js con el código estén disponibles.

Nota: Debido a la construcción de Angular Material y a la orientación a componentes de este módulo, ten en cuenta el requisito de contar con la librería AngularJS al menos en su versión 1.4.8. Se recomienda en el momento de escribir este artículo AngularJS 1.5.

Usar Angular Material

Una vez cargado el CSS y el Javascript necesario para usar Angular Material tenemos a nuestra disposición una serie enorme de elementos para producir de una manera cómoda diseños basados en Material Design.

Ahora se trata simplemente de estudiar la documentación y revisar todas y cada una de las utilidades que nos ofrece, componentes, clases, y directivas en general. El lenguaje de etiquetas HTML nuevas que implementan (lo que llamamos componentes), atributos que se pueden aplicar a etiquetas existentes y sus valores es muy rico y quizás cuando estamos empezando pueda resultar un poco farragoso, pero poco a poco es factible familiarizarse con todos estos elementos e incorporarlos a los proyectos.

En la propia página de Material Design se ofrecen generalmente demos de uso de los componentes, donde podemos ver el código para implementarlos. Esas implementaciones sugeridas las podemos alterar y personalizar, agregando nuevos elementos. Generalmente es una buena opción copiar el código de un componente y personalizarlo ya para nuestro caso.

Por ejemplo, un botón se consigue:

```
<md-button>Texto</md-button>
```

Pero también podría venir con un icono:

```
<md-button class="md-fab md-primary" aria-label="Use Android">
```

```
<md-icon md-svg-src="img/icons/android.svg"></md-icon>  
</md-button>
```

Hay componentes además que también nos soportan comportamientos, como es el caso de las cajas de diálogo. En este caso habrá también que documentarse para ver cómo se pueden producir no solo con las etiquetas necesarias para incluirlas en el HTML de la página, sino también con el Javascript necesario para ponerlas en funcionamiento o los eventos que produzcan cuando el usuario interaccione con los componentes.

En si el proyecto es muy amplio, pero merece la pena el esfuerzo de estudiarlo. Entender y dominar sus componentes llevará su tiempo, pero a la larga nos ahorrará mucho trabajo del lado del diseño y nos asegura un nivel de calidad estético y una usabilidad elevada en nuestras aplicaciones.

Acabamos esta review agradeciendo a Alberto Basalo sus clases en el [Curso de Front Edge de EscuelaIT](#), donde hemos podido realizar una aplicación AngularJS con Angular Material, lo que sin duda facilita enormemente comenzar a usar esta librería y suavizar enormemente la curva de aprendizaje.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 16/02/2016
Disponible online en <http://desarrolloweb.com/articulos/angular-material.html>

Components en Angular 1.5

Tratamos el desarrollo basado en componentes de AngularJS, arquitecturas de aplicaciones con componentes personalizados. Cómo crear componentes, definir la lógica con los componentes, la interoperabilidad para comunicación entre components, etc.

Componentes en AngularJS 1.5

Desarrollo de components en Angular 1.5, en esta guía a la componetización aprenderás a realizar realizar tus propios componentes.

En el [Manual de AngularJS](#) no habíamos tratado todavía un asunto tan interesante y novedoso como es la creación de componentes, una nueva manera de realizar el tipo de trabajo que antes venía realizándose con las directivas. Hoy la alternativa de los componentes está conviviendo con las propias directivas, en la versión 1.x, aunque se prevé que las directivas finalicen definitivamente, y se sustituirán completamente por componentes en el nuevo Angular 2.0. Esto es debido a que los componentes y las directivas sirven más o menos para hacer las mismas cosas, extender el HTML con nuevas funcionalidades que se puedan implementar de manera declarativa. Sin embargo la orientación a componentes permite una mayor sencillez y versatilidad para el desarrollo.

En este artículo estudiaremos algunas diferencias entre componentes y directivas y veremos cómo realizar un componente sencillo, de modo que nuestra aplicación Angular sea capaz de conocer nuevas etiquetas de HTML. Usaremos la versión 1.5 de AngularJS, aunque desde hace menos tiempo han actualizado ramas anteriores de AngularJS (versión 1.3 en adelante) para que también admita desarrollo basado en componentes.



Directivas Vs componentes

Las directivas de AngularJS, así como los componentes se crean añadiendo la funcionalidad a un módulo (module) igual que otros elementos como controladores o factorías.

La diferencia fundamental es que, mientras que las directivas recibían una función, igual que controllers o factories, los componentes reciben un objeto. Realmente es parecido en el sentido en el que las directivas al final debían devolver un objeto y los componentes directamente sirven ese objeto al método `component()`.

Directivas se declaraban así:

```
.directive('nombre_directiva', function() {  
  return {  
    //...  
  };  
});
```

Los componentes por su parte se declaran:

```
.component('nombre_componente', {  
  //...  
});
```

Este es uno de los muchos detalles, puesto que muchos otros conceptos han sido simplificados con respecto a las directivas.

Pero en la práctica los componentes además nos permiten extender el HTML de otra manera, creando nuevas etiquetas completas que podremos incorporar y que esconden dentro sus propias complejidades, permitiendo una mayor reutilización y manteniendo encapsulada su complejidad.

Esto nos permite simplificar el marcado de las vistas. Si antes teníamos algo como:

```
<h2>Contactos</h2>  
<ul class="lista">  
  <li ng-repeat="elem in vm.contactos">  
    {{elem.nombre}} <span>{{elem.telefono}}</span>  
  </li>  
</ul>
```

Ahora en resumen podremos tener algo como:

```
<contactos items="vm.contactos"></contactos>
```

El componente `contactos` encapsula todo ese marcado y a su vez podría estar usando otros componentes para encapsular cada uno de los contactos individuales, ya que unos componentes se apoyan en otros para resolver los problemas.

Si tenemos varias listas de contactos simplemente le pasamos un juego de datos distinto y listo. Si tenemos otro proyecto que maneje contactos, se hace mucho más cómodo reutilizar el componente. También permite aislar mejor los distintos componentes del proyecto entero, el scope y realizar testing de cada

componente de manera independiente, entre otras ventajas con respecto a las directivas.

Crear componentes básicos

De momento vamos a aprender a crear componentes que no trabajan con datos, que simplemente encapsulan un marcado HTML, que ya es bastante. Así podremos aprender de una manera sencilla a crear componentes, incluirlos en nuestra aplicación y usarlos desde las vistas.

Simplificaremos al máximo el ejemplo, teniendo en cuenta que generalmente para facilitar la reutilización y la organización de archivos del proyecto, lo ideal sería colocar el componente en un archivo aparte, incluso su HTML separarlo a otro fichero independiente. Veremos aún en este artículo varias de estas mejoras.

Este es el Javascript de nuestro componente:

```
angular
  .module("appProbandoComponent", [])
  .component("probandoComponent", {
    template: [
      '<div class="miClase">',
      '<span>Esto es algo de HTML en el componente</span> ',
      '<b>Angular Rules!</b>',
      '</div>'
    ].join('')
  });
```

Lo importante que tienes que observar del código anterior:

- Usamos el método `component()` para definir el componente. Ese método recibe el nombre del componente y el objeto que lo define.
- El nombre del componente que pasamos a la función `component` debe estar escrito en "camel case", si es que está compuesto por varias palabras. Pero luego veremos que en el HTML nos referimos al componente separando las palabras por guiones.
- En el objeto enviado al componente le indicamos el atributo "template" que tiene el HTML del marcado del componente. Luego veremos que existe un atributo llamado "templateUrl" que es más interesante por contener una ruta para un archivo HTML donde colocaremos el marcado de manera separada.

Y ahora veamos cómo se usaría:

```
<probando-component></probando-component>
```

Como ves es como si hubiésemos creado una nueva etiqueta, que el navegador entenderá perfectamente gracias a AngularJS. No obstante, por la construcción de nuestro ejemplo, donde estamos usando la inicialización básica de Angular, vamos a necesitar el `ng-app` en la etiqueta BODY o HTML. Mira el código html completo de nuestro ejemplo.

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Probando componentes</title>
</head>
<body ng-app="appProbandoComponent">
  <probando-component></probando-component>

  <script src="bower_components/angular/angular.min.js"></script>
  <script>
    angular
      .module("appProbandoComponent", [])
      .component("probandoComponent", {
        template: [
          '<div class="mi clase">',
          '<span>Esto es algo de HTML en el componente</span> ',
          '<b>Angular Rules!</b>',
          '</div>'
        ].join('')
      });
  </script>
</body>
</html>
```

Organizar los archivos de un componente

Repetimos, que no hemos usado muy buenas prácticas en este ejercicio, por simplificar las cosas y centrarnos simplemente en el método `component()` que nos permite crear el componente. Así que vamos a detenernos ahora en mejorar lo presente. La idea es estructurar los componentes en una carpeta específica del proyecto, además de separar el contenido de la presentación, etc.

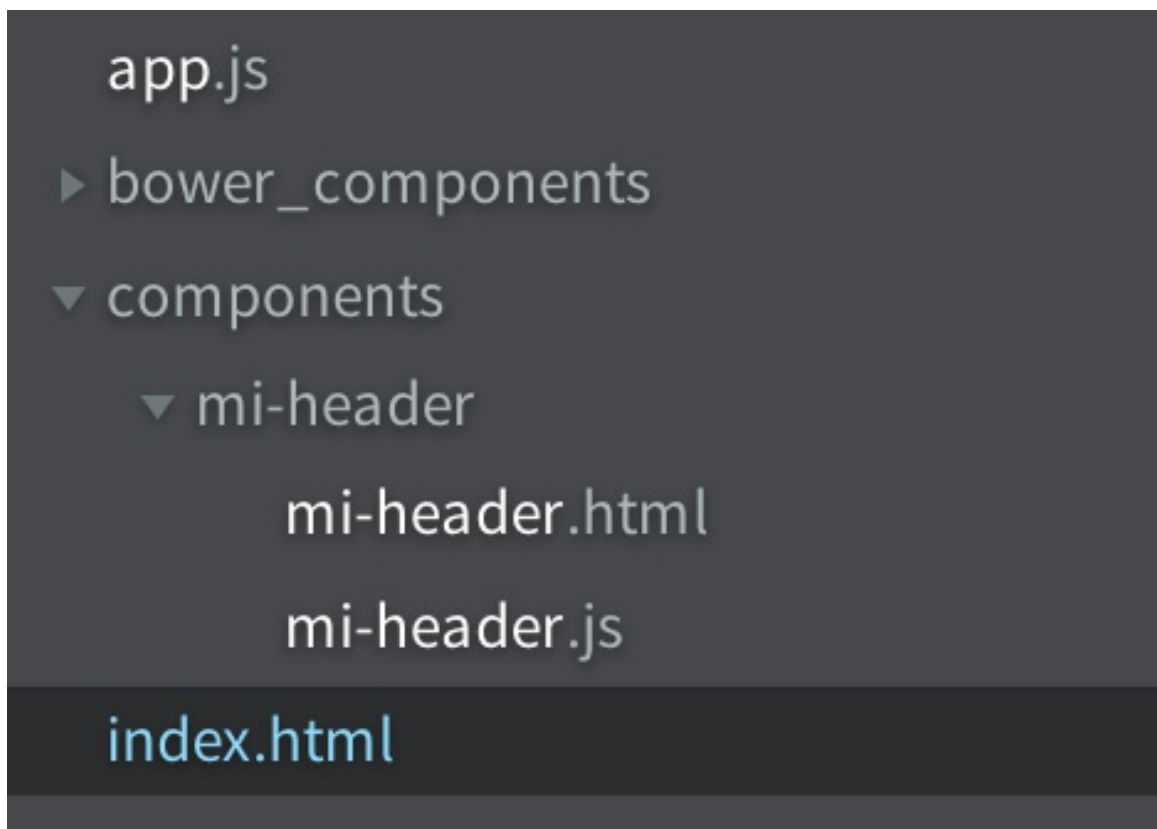
Carpeta de componentes:

Como primera buena práctica, vamos a crear una carpeta donde colocaremos todos los componentes que vayamos desarrollando para nuestra aplicación.

A esta carpeta la llamaremos "components" y a su vez colocaremos dentro de ella cada componente en un directorio independiente.

Nuestro componente ahora se llama "mi-header". Dentro de nuestro proyecto lo colocaremos en la carpeta "componentes/mi-header". Dentro de esa carpeta vamos a crear dos archivos distintos, uno con el Javascript del componente y otro con el HTML, de modo que separaremos el contenido de la lógica.

En la siguiente imagen encuentras la propuesta de organización de archivos.



El Javascript del componente:

Ahora veamos nuestro Javascript para definir el componente. Lo más importante aquí es que hemos usado un módulo independiente para este componente. Sobre este "module" crearemos el componente en sí con el método `component()`. Lo interesante de esta alternativa es que para reutilizar el componente en otro proyecto será simplemente traernos este módulo y declararlo como dependencia.

Este sería nuestro archivo `mi-header.js`

```
angular.module("miHeader", [])
  .component("miHeader", {
    templateUrl: "../components/mi-header/mi-header.html"
  });
```

En esta ocasión comprobarás que en lugar del marcado del componente, definido con el atributo "template" que vimos antes, tenemos el atributo "templateUrl" donde indicamos la ruta del HTML del componente, desde la raíz de nuestro proyecto.

Marcado del componente:

El código HTML del componente ahora lo tenemos en un archivo `.html`, lo que nos ofrece muchas ventajas de cara al mantenimiento, como te imaginarás.

Este sería el código HTML de nuestro archivo `mi-header.html`

```
<header>
  <h1>Esta es mi cabecera</h1>
  <div>Controles de cabecera</div>
</header>
```

Declaramos la dependencia del componente en el módulo principal de la aplicación:

Lo hemos dicho antes, al estar el componente en un módulo independiente, en nuestro módulo principal de aplicación debemos declarar que estamos dependiendo de este componente.

Esto es lo que tenemos en app.js, que sería el Javascript principal de nuestra aplicación. Está vacío, salvo porque hemos declarado la dependencia con el módulo llamado "mi-header".

```
angular
  .module("appProbandoComponent", ["miHeader"])
```

Usamos el componente desde el index.html del proyecto:

Ahora nos queda usar el componente. Esto lo hacemos en el index.html, colocando la etiqueta del componente. Pero además recuerda definir el ng-app con el nombre del módulo principal de la aplicación, así como incluir los diferentes scripts.

Veamos el código completo de index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Probando componentes</title>
</head>
<body ng-app="appProbandoComponent">
  <mi-header></mi-header>

  <script src="bower_components/angular/angular.min.js"></script>
  <script src="components/mi-header/mi-header.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

Todo este código también dependerá del gestor de dependencias que uses y el tipo de arranque que prefieras en tu aplicación Angular. Pero repara que para usar el componente simplemente colocamos la etiqueta HTML "mi-header".

```
<mi-header></mi-header>
```

De momento eso es todo. Seguramente podrás darte cuenta de las ventajas de la componetización en AngularJS. Ahora es empezar a usarla. No obstante, seguiremos informando sobre cómo realizar otras cosas con componentes, porque hasta ahora con lo que has visto solo son como "includes" de código HTML, y realmente podemos derivar mucha lógica de nuestra aplicación dentro de los componentes.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 22/02/2016
Disponible online en <http://desarrolloweb.com/articulos/componentes-angularjs-15.html>

Controladores en components de Angular

Los componentes en AngularJS mantienen la lógica en controladores independientes, que debemos definir durante la declaración component.

El desarrollo en Angular ahora está basado en componentes, una tendencia que también encontraremos en la versión 2.0 del framework, por lo que conviene ir familiarizándose con ella. Por otra parte nos ofrece diversas ventajas y hasta simplicidad con respecto al desarrollo con directivas, por lo que nos interesará invertir algo de tiempo en ponernos al día.

Si quieres saber cómo trabajar con componentes en Angular y empiezas desde cero, te recomendamos comenzar por la lectura de un artículo anterior donde se trataron las [bases del desarrollo basado en componentes](#). En este artículo vamos a explicar cómo incorporar lógica (programación) en un componente, permitiendo la posibilidad de realizar cálculos o responder a acciones e interacción por parte del usuario.



Controller en la declaración component

Como ya estamos acostumbrados en general cuando usamos Angular, los controladores son el lugar donde comenzamos a aplicar lógica de la aplicación, aunque también sabemos que muchas veces éstos apoyan parte de su comportamiento en otro tipo de artefactos como factorías y servicios.

En el caso de componentes todo sigue igual. Tendremos un controlador que nos permitirá escribir el código a ejecutar para la inicialización de cada instancia del componente y para definir comportamientos ante diversas situaciones.

El controlador se debe definir al declarar el componente. Ya sabes que para crear el componente usas la función de angular "component". A esa función le pasamos un objeto con diversos datos, como el template HTML del componente. [Todo eso se vió en el anterior artículo.](#)

Para definir el controlador, al objeto de declaración component, le pasamos una nueva propiedad llamada "controller". En esa propiedad colocamos una función que será nuestro propio controlador.

Podemos escribir directamente el código de nuestro controlador con una función anónima, tal como ves en el siguiente código:

```
angular.module('cuentaClics', [])
.component('cuentaClics', {
  templateUrl: './components/cuenta-clics/cuenta-clics.html',
  controller: function(){
    var vm = this;
    vm.numClics = 0;
    vm.incrementaClic = function(){
      vm.numClics ++;
    }
  }
});
```

También como alternativa es posible indicar el nombre de un controlador que haya definido en éste u otro módulo. Por tanto el código que ves a continuación tendrá el exactamente el mismo efecto:

```
angular.module('cuentaClics', [])
.controller('CuentaClicsController', function(){
  var vm = this;
  vm.numClics = 0;
  vm.incrementaClic = function(){
    vm.numClics ++;
  }
})
.component('cuentaClics', {
  templateUrl: './components/cuenta-clics/cuenta-clics.html',
  controller: 'CuentaClicsController'
});
```

Como has comprobado en este sencillo controlador tenemos las dos cosas que se suelen encontrar en ellos, datos como es el caso de "numClics" y funciones para realizar comportamientos, como es el caso de "incrementaClic".

Acceso a los controladores dentro del HTML de un componente

Ahora tenemos que ver cómo podemos acceder a las propiedades de un controlador dentro del template de un componente, ya sea para visualizar uno de sus datos o invocar alguno de sus métodos.

Ésto se realiza tal como venimos operando con controladores en el pasado, por medio de una variable que hace referencia al controlador y mediante la cual se accede a sus propiedades y métodos. La única cosa que debemos saber es que en components de AngularJS los controladores tienen el nombre de \$ctrl dentro de las vistas.

Nota: Con el método de trabajo que se conoce con "Controller As" éramos capaces de definir diversos nombres para conocer a los controladores dentro de diversos scopes que puede tener una aplicación, de modo que se puedan anidar controladores y no tener problemas para acceder a datos o métodos de controladores específicos, incluso cuando existen propiedades y valores que tienen los mismos nombres. En controllers de components también podemos asignar nombres para definir cómo se debe de conocer al controlador dentro del HTML del componente, usando la propiedad "controllerAs":

```
.component('cuentaClics', {  
  templateUrl: './components/cuenta-clics/cuenta-clics.html',  
  controller: 'CuentaClicsController',  
  controllerAs: 'vm'  
});
```

Sin embargo en principio no necesitaríamos cambiar nunca ese nombre de controlador, ya que la arquitectura de componentes hace que los datos ofrecidos por un controlador estén encapsulados dentro de ese componente. Es decir, en componentes hijo no somos capaces de acceder a los datos de los controladores de los padres u otros antecesores y por ello no es necesario el cambio del nombre del controlador con "controllerAs". Solo para aclarar posibles dudas cabe adelantar que el mecanismo para compartir datos de unos componentes a otros es el bindeo, enviando datos a los componentes hijos que los puedan necesitar. Esas técnicas de binding en componentes son las que vamos a tratar en el artículo siguiente.

En este HTML podrás encontrar el uso de la propiedad y método que habíamos definido en el controlador de nuestro componente cuentaClics.

```
<div>  
  <h2>Cuenta Clics</h2>  
  <p>Hasta el momento has hecho {{ctrl.numClics}}</p>  
  <button ng-click="$ctrl.incrementaClic()">Haz clic para incrementar el contador</button>  
</div>
```

Usar un componente

En este momento asumimos que la mayoría conocerá el flujo para usar un componente como este, que hemos definido en un módulo independiente. Si es tu caso puedes saltarte esta sección, pero si estás comenzando con Angular o componentes quizás te vengan bien las siguientes pautas.

Nota: Nuestro componente es bueno colocarlo en un módulo independiente, donde solo esté este componente, para que podamos reutilizarlo fácilmente entre distintos proyectos, o incluso para que lo podamos compartir con otras personas.

1.- Definimos el componente, normalmente en una carpeta independiente donde estarán todos los ficheros. De momento tendremos el .js de la declaración del componente, en su módulo independiente, y el .html del template.

2.- Incluimos el script donde está el componente (el .js de la declaración). Esto lo podrás realizar de diversas formas, donde la más sencilla sería colocar la correspondiente etiqueta SCRIPT con el src hacia la ruta del componente.

```
<script src="components/cuenta-clics/cuenta-clics.js"></script>
```

3.- Al crear el módulo general de nuestra aplicación, definimos la dependencia con el módulo donde hemos creado nuestro component.

```
angular.module('miapp', [ 'cuentaClics' ])
```

4.- Luego ya solo queda usar el componente, las veces que sean necesarias, usando la etiqueta correspondiente con el nombre del componente que hayamos creado. Solo ten en cuenta que si el nombre del componente en Javascript tiene notación "camel case", en el HTML tenemos que separar las palabras por guiones.

```
<cuenta-clics></cuenta-clics>
```

Eso es todo, tu componente debería funcionar sin problemas. Si no es así, revisa que la ruta en el src del script esté correcta, que el nombre del módulo donde está el component esté correcto, tenga notación camel case para separar palabras y que esté definida la dependencia en el módulo de tu aplicación. Revisa por último que la etiqueta HTML tiene separadas las palabras por guiones.

En la siguiente entrega de esta serie dedicada a componentes en Angular veremos algo muy interesante como es el binding entre componentes para facilitar la interoperabilidad.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 28/03/2016
Disponible online en <http://desarrolloweb.com/articulos/controladores-componentes-angular.html>

Bindings entre componentes AngularJS 1.5

Cómo compartir datos entre unos componentes y otros, a fin de facilitar la interoperabilidad, por medio de mecanismos de binding, en AngularJS 1.5.

En los últimos artículos del [Manual de AngularJS](#) hemos explicado las [bases del desarrollo basado en componentes](#) y [cómo funcionan los controladores en componentes](#). Pero todavía tenemos que aprender más cosas para lanzarnos a la arquitectura de aplicación basada en componentes.

En esta ocasión vamos a explicar cómo pueden unos componentes interoperar con otros, por medio del paso de datos o información para su funcionamiento. Para ello usamos mecanismos conocidos como el bindeo. Para quien ya esté familiarizado con el desarrollo de directivas seguramente le resultará todo muy familiar, aunque el que venga sin ese conocimiento previo tampoco ha de preocuparse porque es todo

bastante sencillo.



Arquitectura de componentes

Cada componente es responsable por hacer una parte del trabajo de la aplicación, pero para trabajar no están solos, sino que deben relacionarse y comunicarse unos con otros.

Bajo la arquitectura de componentes tenemos un árbol de elementos. Forman una estructura donde unos componentes incluyen a otros, donde cada uno de ellos encapsula un HTML y cierta lógica de aplicación.

La clave es que cada componente tiene su scope (ámbito) independiente y único. Un componente solo conoce aquellos datos que su controlador maneja, no pudiendo acceder a los scope de otros componentes, ya sean padres o hijos. Esa estructura facilita el desarrollo de aplicaciones, porque uno puede saber a priori dónde está el código que afecta a cada parte de la aplicación.

Esto se ve mejor en contraposición con una aplicación tradicional de Angular:

- En una aplicación de Angular, desde cualquier componente del código podemos afectar a cualquier parte de la aplicación, ya que el ámbito se puede compartir por mecanismos como la herencia de scopes y los observadores. Es una situación práctica que a veces deriva en problemas al entender las aplicaciones y al depurarlas.
- En una aplicación de componentes cada componente afecta al HTML que tiene dentro y puede modificar los datos que se están mostrando dentro de ese HTML. Eso hace fácil localizar el código que afecta a los datos de cualquier parte de la aplicación.

Para que componentes independientes puedan colaborar entre sí, de modo que consigamos entre todos una aplicación, es necesario que se intercambien datos. Esto se hace por medio de mecanismos de binding y se pueden producir a varios niveles.

Cómo se pasan los datos a los componentes

En la arquitectura de componentes los datos se pasan por medio de propiedades en el HTML y para que funcione se tiene que realizar la declaración de los datos compartidos en el JS.

Las propiedades del HTML funcionan como modificadores de los componentes, enviando datos que pueden afectar al componente de diversas maneras.

```
<mi-componente dato="valor"></mi-componente>
```

En el anterior HTML tenemos una propiedad llamada "dato" que dentro del componente equivale a "valor".

En el Javascript del componente, en su definición, se debe declarar qué propiedades se pueden recibir y qué tipos de binding se van a implementar sobre ellas. Para ello se debe declarar cada una de las propiedades en el objeto "bindings", indicando un símbolo que aclare qué tipo de bindeo se va a realizar.

```
angular.module("componenteBinding", [])  
.component("componenteBinding", {  
  templateUrl: "../js/components/componente-binding/componente-binding.html",  
  controller: "componenteBindingController",  
  bindings: {  
    user: "=",  
    num: "@"  
  }  
});
```

En el código anterior se está declarando un binding con dos atributos, "user" y "num".

Tipos de bindeo


El bindeo entre componentes puede ser de varios tipos. Podemos tener binding a dos direcciones, algo típico de Angular, donde los datos viajan desde un componente al otro y desde el segundo al primero. Pero ahora en Angular 1.5 también podemos crear binding en una sola dirección, así como también se puede realizar envío de datos en crudo, que una vez se entregan no quedan bindeados ni en una dirección ni en otra.

En la declaración bindings, cada tipo de bindeo se define por medio de un símbolo, los principales son:

"=": esto sirve para entregar una referencia a un objeto. Por tanto cualquier cosa que se esté entregando se comparte entre componentes.

"@": esto sirve para entregar un valor. No existe binding de ningún tipo.

"<": (Símbolo "menor que"): esto sirve para bindear en una única dirección, o 1 way binding. El padre le transfiere al hijo un valor, pero aunque lo modifique el hijo el nuevo valor no viaja al padre. Sin embargo, si el padre lo cambia, sí se cambia en el hijo.

Nota: Pero ojo, con la alternativa "1 way binding" (""), si transfieres un objeto o un array, al pasarle la referencia en la práctica todavía estarás produciendo un 2 way binding tradicional de Angular.

"&": esta última alternativa permite enviar un puntero a una función.

Ejemplo de bindeo en componentes

Antes de terminar vamos a ver un ejemplo para ilustrar todo lo que hemos aprendido sobre componentes a lo largo de los anteriores artículos. Es un ejemplo sencillo pero práctico. Son dos componentes que se transfieren información para completar un objetivo común, que es crear un listado de usuarios.

Tendremos un componente padre que es el listado completo de usuarios y un componente hijo que es cada uno de los ítem del listado (el detalle de un usuario único). Por decirlo de otro modo, el componente padre producirá una repetición y cada uno de los ítem de esa repetición será implementado por un componente hijo.

Hemos llamado a nuestros componentes "user-list" y "user-detail". Cada componente a su vez tiene dos archivos de código, uno el .js, con el registro del componente en sí, y otro .html con el HTML local de ese componente.

Comenzamos viendo el **código de mi listado de usuarios: user-list**

El archivo user-list.js

```
angular.module("userList", [])
.component("userList", {
  templateUrl: "../js/components/user-list/user-list.html",
  controller: function($http){
    var vm = this;

    $http.get("http://jsonplaceholder.typicode.com/users")
    .then(function(respuesta){
      vm.usuarios = respuesta.data;
    })
  }
});
```

Como puedes ver, me traigo el listado de usuarios de un servicio REST. Ese listado de usuarios lo vuelco en la variable vm.usuarios, que es el único dato que este controlador ofrece a la vista.

Código de user-list.html

```
<user-detail ng-repeat="item in $ctrl.usuarios" usuario="item" numero="{{ $index }}"></user-detail>
```

El código HTML es un único componente que tiene un ng-repeat, por lo que se produce la iteración por todos los usuarios. Cada usuario se muestra con el componente user-detail. A ese componente le estamos pasando dos bindings, cuyos valores se entregan por atributos del HTML.

Es importante reparar la diferencia entre el atributo usuario="item" y numero="{{ \$index }}". La diferencia de las dos llaves es que {{ \$index }} lo vamos a pasar por valor, no por referencia. Lo confirmarás en el siguiente listado.

Código de user-detail.js

```
angular.module("userDetail", [])
.component("userDetail", {
  templateUrl: "../js/components/user-detail/user-detail.html",
  controller: function(){
    var vm = this;
    vm.cambiarEmail = function(){
      vm.usuario.email = "miguel@desarrolloweb.com";
    }
  },
  bindings: {
    usuario: "=", //ten en cuenta que si usaras una propiedad con nombre camelCase (ejemplo: usuarioIndividual), cuando la utilices como atributo de la etiqueta deberás escribirla sin mayúsculas y con guiones: usuario-individual
    numero: "@"
  }
})
```

Aquí la novedad está en "bindings", donde estamos declarando los dos bindeos que este componente espera recibir. Ahora lo importante es:

- El campo "usuario" lo recibe como bindeo. Es un objeto, pasado por referencia. En ese caso se produce un 2 way binding.
- El campo "numero" lo recibe como valor. Le llega solo un dato que, por mucho que se cambie en el componente, no se transfiere nada al padre.

Además fíjate que el controlador casi no tiene código. Solo hemos definido una función para cambiar un dato el objeto usuario, de modo que se compruebe si se produce el binding hacia arriba.

En la vista podremos usar los bindeos como si fueran datos proporcionados por el propio controlador y en el controlador también los podremos usar como si fueran propiedades suyas normales. Esto lo comprobamos con el siguiente listado.

Archivo user-detail.html

```
<div class="user" ng-click="$ctrl.cambiarEmail()">
  <h2>{{ $ctrl.numero }}</h2>
  Nombre: {{ $ctrl.usuario.name }}
  <br>
  Email: {{ $ctrl.usuario.email }}
</div>
```

Aprecia como los datos que me vienen por el binding los trato como si fueran del controlador: `$ctrl.numero` y `$ctrl.usuario`

Puedes encontrar el [código de estos ejemplos de componentes en Angular 1.5 en Github](#).

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en *31/03/2016*

Disponible online en <http://desarrolloweb.com/articulos/bindings-componentes-angularjs.html>